EViews Database Extension Interface

September 23, 2014

Table of Contents

Introduction[1]

Up until now, EViews users could only open a handful of EViews supported foreign database formats (such as DataStream, EcoWin, Haver, FRED, etc.) directly within EViews. If their data resided in an unsupported database format, users were limited to using ODBC (if an ODBC driver was available) or using an intermediate file format (such as XLS, CSV or HTML) or the Windows clipboard to exchange data. There are several limitations to this approach:

- working with generic formats such as text and Excel files can be complicated since these formats are not self-describing so that additional information about the structure of the files may be needed for EViews to understand the files.
- frequency information must be inferred by EViews from date identifiers accompanying the data and this is not always reliable.
- there is no way to communicate additional attributes along with the observation values such as source, units, etc.
- data brought into EViews this way cannot be "linked" back to the source to allow for automatic refreshes when a workfile is loaded. (Data can only be updated by rerunning an EViews program to manually perform the refresh.)

In an effort to provide EViews with access to as broad a range of data sources as possible, we have created a new Database Extension Interface. Now, any external data source that implements this interface can be opened directly from within EViews and used just like an EViews database.

By implementing a database extension for an external database format, you can extend EViews to include one or more of the following functionalities:

- an EViews user can ask what objects the external database contains
- an EViews user can read data objects from the external database
- an EViews user can write data objects to the external database
- an EViews user can browse through the contents of the external database using a custom graphical interface developed specifically for the data source

The Database Extension Interface is a set of COM interfaces. Supporting a new format involves creating a small library (usually housed in a DLL) that contains COM objects that implement these interfaces. The library can be developed in many different programming environments including native C++ and Microsoft .NET. These COM objects are used by EViews to interact with the underlying database.

The two main EViews COM interfaces for EDX are:

```
IDatabaseManager
IDatabase
```

[1] All product names mentioned may be trademarks or registered trademarks of their respective companies

IDatabaseManager is the initial contact point between EViews and the custom database format. A single database manager is created once per session for each format. The manager provides EViews with information about the database format (e.g. attributes such as format name, description, and whether the format is file or server based). EViews uses the manager to open a connection to a server if necessary, and create, rename, copy, or delete databases.

IDatabase represents a currently open database. EViews calls functions in IDatabase whenever a user needs to search the database or when a user tries to read or write a data object to the database (such as a series object, matrix/vector object, or string). IDatabase also provides additional management functions such as copying, renaming, and deleting objects within the database if the format supports writing.

To help you implement your database extension, EViews also provides several utility classes as part of the API. These classes are provided to help facilitate development of a database extension, but their use is optional. The Frequency class exports a variety of functionality available within EViews for working with calendar date and frequency information. The JsonReader and JsonWriter classes provide functionality to assist with processing content in JavaScript Object Notation (JSON) format.

Note that the EViewsEdx type library that declares these interfaces also contains a class EViewsDatabaseManager. This class is intended for applications where a user would like to work with data stored within EViews proprietary file formats (EViews databases and workfiles) within an external application. This class is not likely to be used when implementing an EViews Database Extension. Please see the EViews Database Objects (EDO) documentation for further discussion.

Most of the information passed between EViews and the Database Extension Interface is transferred in sets of attributes where the attributes closely follow EViews conventions. In the material below we assume basic familiarity with EViews. If any terms are unfamiliar, you should refer to the main EViews documentation for further information.

Examples

We provide three examples which will walk you through the methods necessary for working with the EViews Database Extension Interface. The first example illustrates reading from a single text file (see File Based Database). The second example uses multiple XML files in a folder, and demonstrates both reading and writing (see XML Folder Based Database). Finally, we walk through a Read Only SQL Server database example (see SQL Server Database).

Note that we have also made available an additional project that contains source code for a full production implementation of an EViews Database Extension to support U.S. Energy Information Administration (EIA) data. This project shows off many advanced features of a database extension including a custom browser implementation. Please visit the EViews web site for details.

File Based Database

We'll begin with a read only database extension that allows you to import data from a text file. The text file for this database contains a line of data for each object in the following format:

```
Name|Type|Frequency|Start|Data
```

where the data is comma delimited.

For example, the text file "test.cdb" contains the following content:

```
X|series|A|1950|1,2,3,4,5,6,7,8,9,10
Y|series|A|1950|11,12,13,14,15,16,17,18,19,20
Z|alpha|A|1950|a,b,c,d,e,f,g,h,i,j
```

We will use Visual Studio to create a VB.NET Class Library that implements a database extension that can read this file.

The complete source code for this example is provided in the EdxSamples project available at http://www.eviews.com/EViews8/Enterprise/EDXeg.html (for Visual Studio 2012). You can examine and modify this to suit your own needs. You may find it useful to follow the steps below as you read through the discussion so you can better understand how the example project was created.

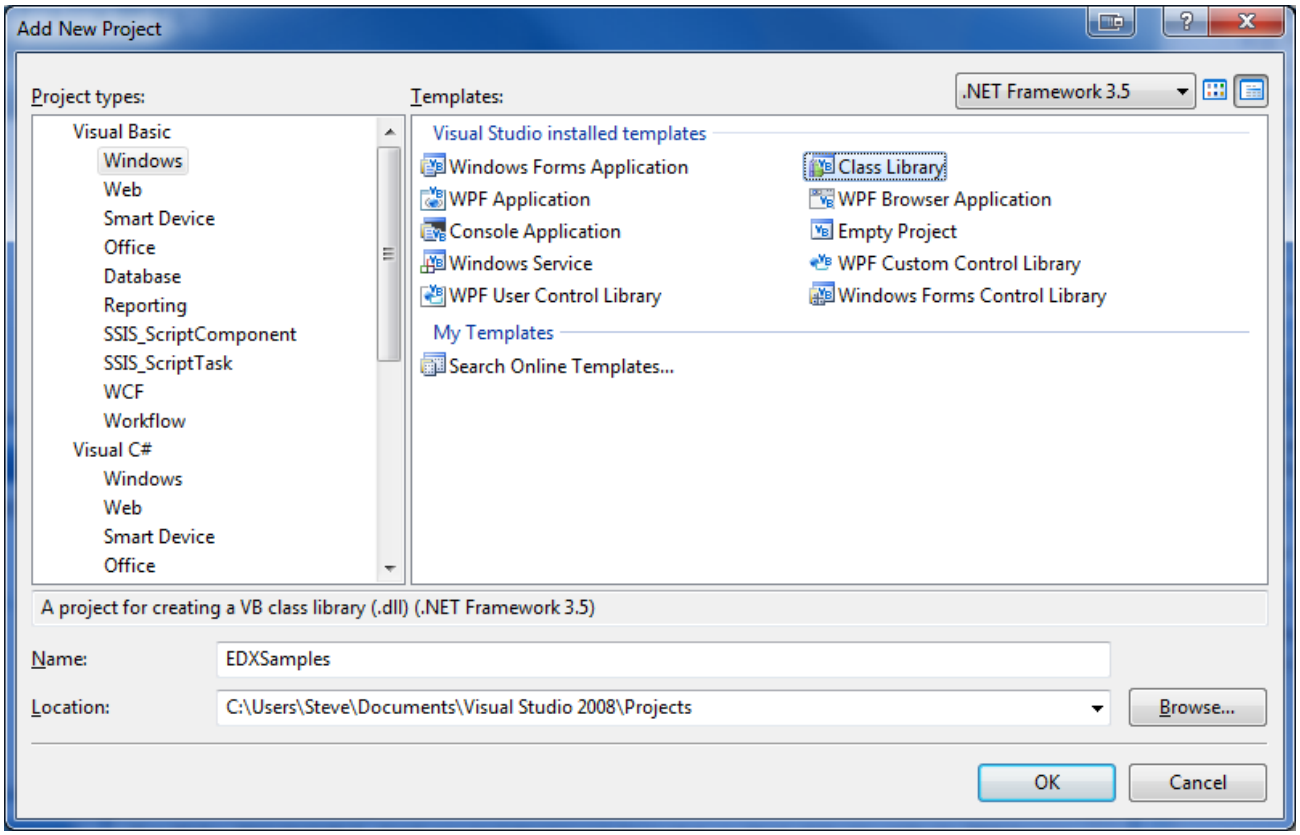
Verify that EViews Database Extension 1.0 Type Library is Registered

Before starting Visual Studio, we need to verify that the EViews Database Extension type library is properly registered on your system. This should happen automatically during the installation of EViews.

To verify this, run EViews, and enter the `REGCOMPONENTS` command. Verify that it says "Database Extension Interface: Registered" in the dialog. If not, click "Yes" to register it on your system.

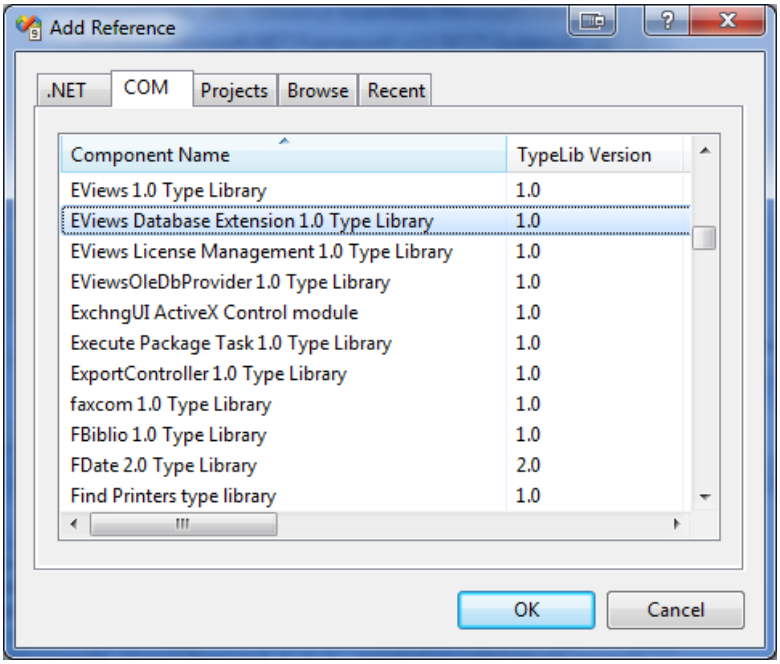
Create a VB Class Library Project

Startup Visual Studio and create a new Visual Basic Class Library project named "EdxSamples".



Add a Reference to the EViewsEdx Type Library

This project will need a reference to the EViews Database Extension type library in order to see our interface definitions. Right-click "EdxSamples" in Solution Explorer and select "Add Reference…".

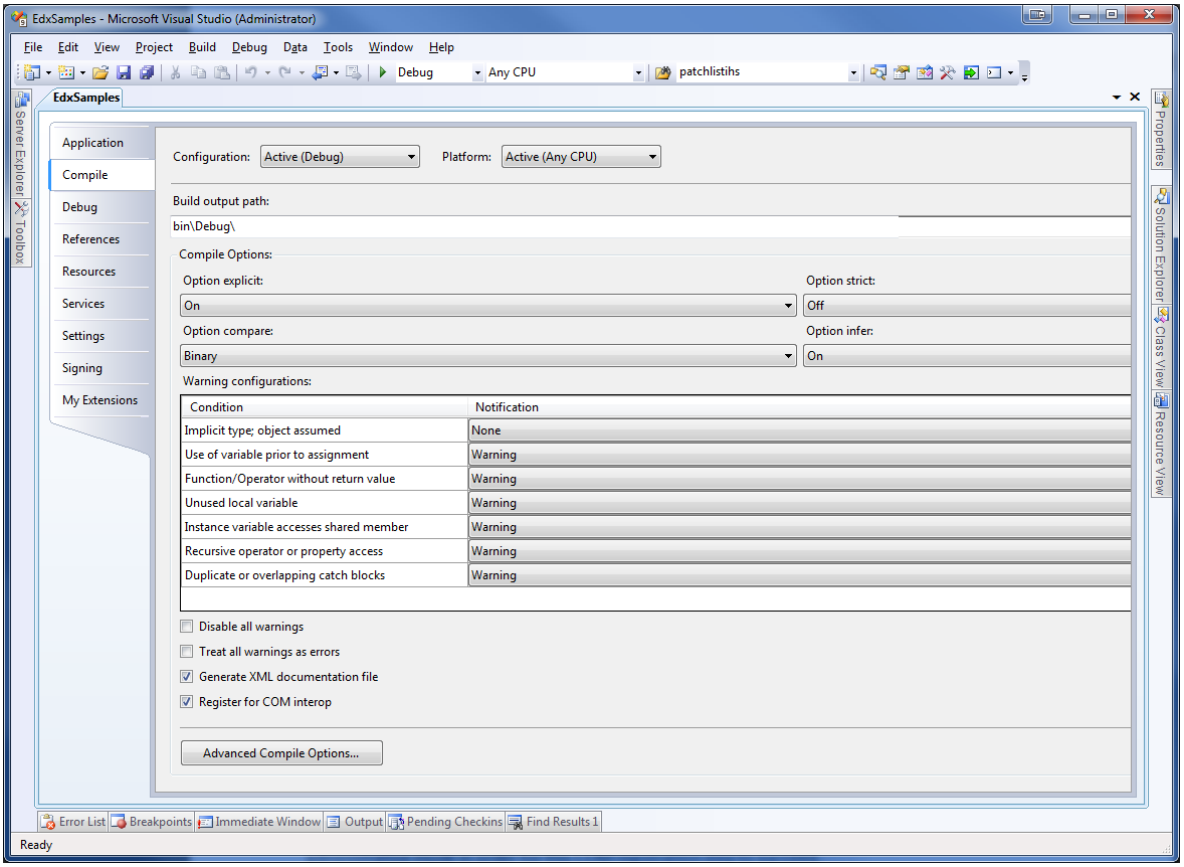


In the Add Reference dialog, select the COM tab and scroll down to "EViews Database Extension 1.0 Type Library" (or "EViews Database Extension 1.0 Beta Type Library" for IHS Beta users). Select this row and click OK. The definitions should now be available inside your project in the namespace "EViewsEdx".

Turn on COM Registration

The COM objects created by our project will need to be registered with Windows before they can be used by EViews. Visual Studio can be configured to do this automatically as part of the project build. To do this, right-click the "EdxSamples" project and select "Properties".

In the left tab bar, select "Compile".



Scroll down to the bottom of this page, and make sure the "Register for COM interop" checkbox is checked. Click the save toolbar button above to save these new settings to the project.

Now, whenever the project is compiled, Visual Studio will register any COM objects in our project with Windows so that they are available to COM clients such as EViews.

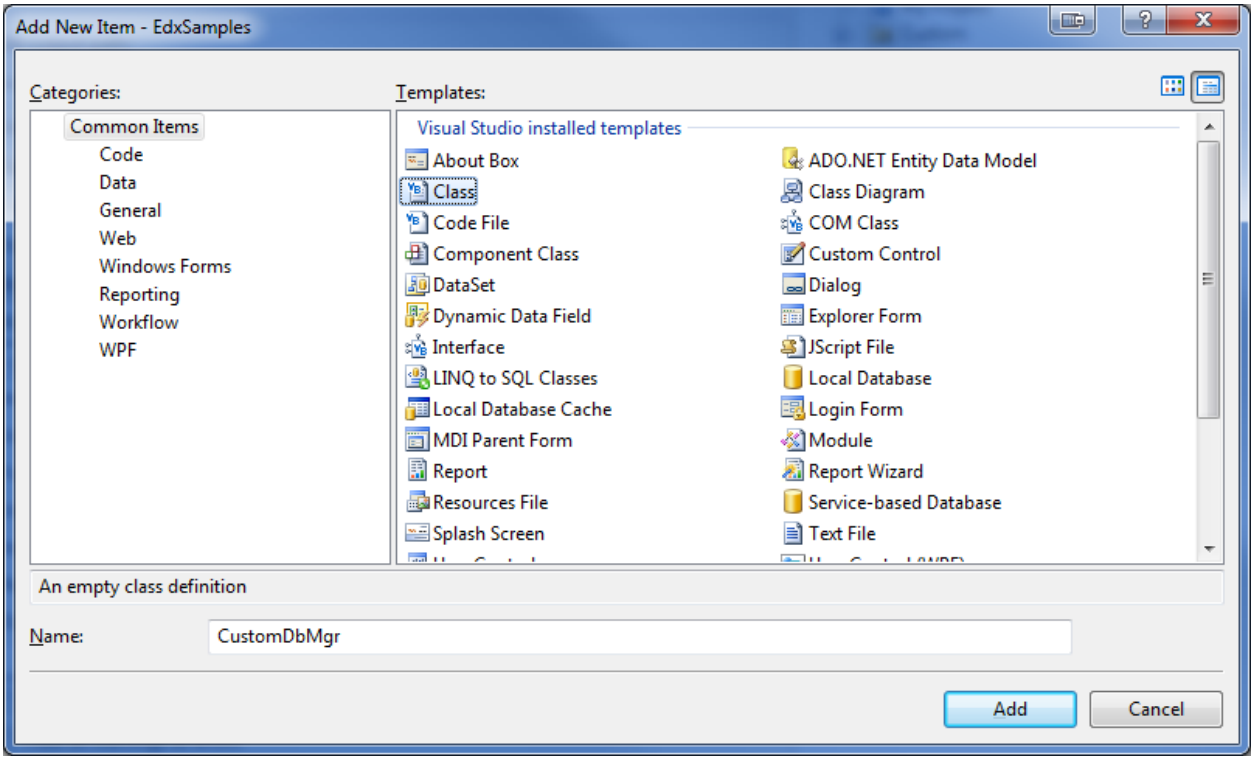
Note that you may need to run Visual Studio in administrative mode in order for this COM registration step to succeed.

Create the Custom folder

Since this .NET project will eventually contain multiple database extension examples, we'll create a new folder named "Custom" to group the Custom Database Extension files together. Right-click the "EdxSamples" project in the Solution Tree view, then select "Add", then "New Folder". Name the folder "Custom".

Create the Database Manager class

Delete the empty Class1.vb file that was previously generated by the Project wizard. Then, right click the "Custom" folder and select "Add", then "Class..." from the menu.



In the Add New Item dialog, select the "Class" template and rename the file to "CustomDbMgr", then click Add.

Since this class will be a public COM object, we need to tell Visual Studio to make this class visible to COM clients and enter its GUID value. We'll also tell Visual Studio not to create a separate COM interface for the class because the only functions that need to be visible over COM are already described in the EViews type libraries. To do this, copy and paste the following lines in the CustomDbMgr.vb file above the "Public Class" line:

```
Imports System.Runtime.InteropServices

<Guid("XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX"), _
 ClassInterface(ClassInterfaceType.None), _
 ComVisible(True)> _
```

```
Public Class CustomDbMgr
```

Note that Visual Studio will automatically assign the ProgId "EdxSamples.CustomDBMgr" to our class. Also, make sure you replace the Guid("XXXX") part with a real GUID. You can generate a new GUID by running the "Create GUID" tool from within Visual Studio (Tools->Create GUID). Make sure you remove any curly braces from the start and end of the GUID string.

Because this class will be our Database Manager, it will need to implement the EViewsEdx.IDatabaseManager interface. Below the "Public Class" line, add the following:

```
Public Class CustomDbMgr
    Implements EViewsEdx.IDatabaseManager
```

After adding this line, .NET will automatically add empty versions of each function that is part of the IDatabaseManager interface to the class.

The first method we'll code is the GetAttributes method.

GetAttributes

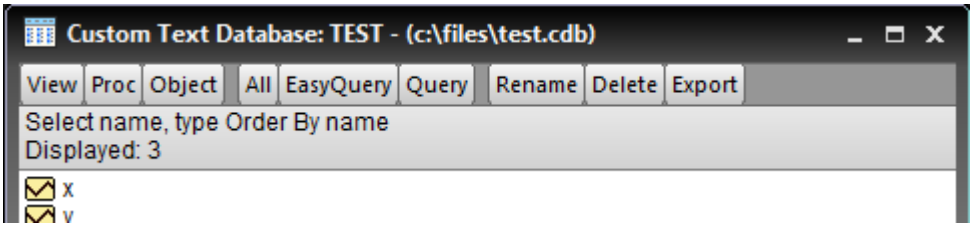
`GetAttributes` returns a list of important attributes about the database format that EViews needs to know to interact with databases of this format. In this example, we'll notify EViews of our format's name, description, type, file extension and search capabilities. We'll also notify EViews that we do not allow create mode and only support read access (no writing):

```
Public Function GetAttributes(ByVal clientInfo As String) As Object _
        Implements EViewsEdx.IDatabaseManager.GetAttributes
    Return "name=CustomDb,description=Custom Text Database," & _
           "type=customdb,ext=cdb,nocreate,readonly,search=all|attr"

End Function
```

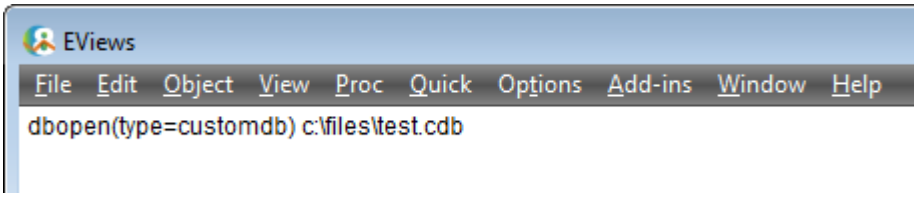
The `clientInfo` parameter that is passed into this method will contain information about the client that instanced the Database Manager. For our purposes, we will ignore this parameter.

The `name` attribute is a short (generally one or two word) name for the format used when error messages are displayed.

The `description` attribute is a longer description for the database format that will be used in EViews dialogs and in the caption for the Database window:



The `type` attribute is used to identify the format in EViews commands that require a type option (e.g. `dbopen(type=customdb)`).



The `ext` attribute notifies EViews that we are a file-based database whose file extension is "cdb".

The `nocreate` attribute notifies EViews that we do not support creating new databases in this format. This will prevent EViews from displaying our database format in the New Database dialog.

The `readonly` attribute notifies EViews that we do not support writing, copying, renaming, or deleting any of our objects in our database.

The `search` attribute notifies EViews what type of searching our database supports. We want both the "All" button (which displays all available objects from the XML folder) and the two "Attr" based search buttons, "EasyQuery" and "Query" (which allow the user to do a search by attributes across objects within the database).

Please refer to Appendix B for details on each of these attributes.

Now that we've told EViews the general details of our database format, our manager needs to be able to return a specific database to EViews.

OpenDb

Whenever an EViews user opens a database in our format, EViews will call the `OpenDb` method on IDatabaseManager to retrieve an IDatabase interface. The IDatabase interface represents a "connection" to our database and will be used by EViews to read from our database.

```
Public Function OpenDb(ByVal databaseId As String, _
                       ByVal oc_mode As EViewsEdx.OpenCreateMode, _
                       ByVal rw_mode As EViewsEdx.ReadWriteMode, _
                       ByVal server As String, _
                       ByVal username As String, _
                       ByVal password As String) As EViewsEdx.IDatabase _
        Implements EViewsEdx.IDatabaseManager.OpenDb
    Return New CustomDb(databaseId)
End Function
```

Before we can use this method, we will need to define the CustomDb class that will be returned by this method.

Create the Database class

Like before, right click the "Custom" folder in the Solution Explorer and select "Add", then "Class…" from the menu. In the Add New Item dialog, select the "Class" template and rename the file to "CustomDb.vb" and click Add.

Unlike CustomDbMgr, this class does not need to be ComVisible as EViews will never instantiate it directly. Instead, CustomDbMgr will create this object and return a reference to it inside the `OpenDb` call.

`CustomDb` will need to implement the EViewsEdx.IDatabase interface in order for it to be usable by EViews:

```
Imports System.Runtime.InteropServices

Public Class CustomDb
    Implements EViewsEdx.IDatabase
```

After adding this line, .NET will automatically add empty versions of each function that is required by our IDatabase interface. We'll begin by adding some class level variables and a new Enumeration:

```
Imports System.Runtime.InteropServices

Public Enum FieldOrder
    Name = 0
    Type
    Freq
    Start
    Data
End Enum

Public Class CustomDb
    Implements EViewsEdx.IDatabase

    Private msDatabaseId As String
    Private miLineIndex As Integer
    Private miLineCount As Integer
    Private maLines() As String
```

and also a new class constructor:

```
Public Sub New(ByVal databaseId As String)
    MyBase.New()

    msDatabaseId = databaseId

    'make sure the file exists and throw an error if it doesn't
    If Not System.IO.File.Exists(msDatabaseId) Then
        Throw New COMException(String.Empty,
EViewsEdx.ErrorCode.FILE_FILENAME_INVALID)
    End If
```

```
    maLines = System.IO.File.ReadAllLines(msDatabaseId)
    miLineCount = UBound(maLines) - LBound(maLines) + 1
End Sub
```

First, we save the `databaseId` value into a member variable so we can refer to it in later function calls. `databaseId` represents the path to the user selected database file. We need to make sure that the specified file exists – if it doesn't, we need to throw the `FILE_FILENAME_INVALID` exception so that EViews knows what to display to the user.

Since our example uses a small text file, we keep things simple by loading the entire text file into memory. We then count the number of lines in the file.

Sequential Searches

When EViews searches through a database to retrieve information about the objects it contains, results are retrieved sequentially (using multiple function calls), not all at once. EViews will first make a call to the database class to initialize the search, then make additional calls to return the results for each object, one at a time. EViews may also abort a search if the user has chosen to cancel the search while the results are still being retrieved.

In our example, since we specified the "all" and "attr" browsing methods in `GetAttributes` (`search=all|attr`), we'll need to fill out the `SearchByAttributes` and `SearchNext` methods.

SearchByAttributes

```
Public Sub SearchByAttributes(ByVal searchExpression As String, _
                              ByVal attrNames As String) _
        Implements EViewsEdx.IDatabase.SearchByAttributes
    'just reset our text array index pointer...
    miLineIndex = 0
End Sub
```

EViews will first call `SearchByAttributes` to allow the database to prepare the list of database objects to return. Then EViews will call `SearchNext` to retrieve the name and attributes of each object in the list until it has retrieved the full list.

Typically, for small databases, `searchExpression` and `attrNames` can be ignored. This is because EViews always performs its own filtering of objects returned during a search so we can simply return every object to EViews and let it do all the filtering work. Large server-based databases may want to limit the number of objects returned to EViews by using the `searchExpression` to select objects and by using `attrNames` to only retrieve attributes that were actually requested by the EViews user (we'll do this later in the Generic SQL Server example).

In our current example, since our database is very small, we'll just ignore these parameters and return everything.

The only thing our function needs to do is to reset the `mLineIndex` value so that the first call to `SearchNext` will always start at the beginning of our array of lines.

SearchNext

```
Public Function SearchNext(ByRef objectId As String, _
                           ByRef attr As Object) As Boolean _
        Implements EViewsEdx.IDatabase.SearchNext
    'check if we're already at the end of the text file...
    If miLineIndex >= miLineCount Then
        Return False
    End If

    'skip to the next non-blank line...
    Do While (maLines(miLineIndex).Length = 0)
        miLineIndex += 1
        If miLineIndex >= miLineCount Then
            'we've reached the end of the text file...
            Return False
        End If
    Loop

    'parse the text line for the object attributes
    'and set the object name
    BuildAttributeString(maLines(miLineIndex), attr, objectId)

    'increment the line pointer to the next line...
    miLineIndex += 1
    Return True
End Function
```

Every time `SearchNext` is called, it retrieves the current line from the text array in memory, then extracts the object name and builds the attribute string (with the help of the BuildAttributeString helper function):

```
Private Sub BuildAttributeString(ByVal vsLine As String, _
                                 ByRef attr As Object, _
                                 Optional ByRef vsName As String = "")
    Dim laObject() As String = Split(vsLine, "|")

    attr = ""

    'name
    vsName = laObject(FieldOrder.Name)

    'freq
    If laObject(FieldOrder.Freq) > "" Then
        attr &= "freq=" & laObject(FieldOrder.Freq)
    End If

    'type
    If laObject(FieldOrder.Type) > "" Then
        If CStr(attr).Length > 0 Then
            attr &= ","
        End If
```

```
        attr &= "type=" & laObject(FieldOrder.Type)
    End If

    'start
    If laObject(FieldOrder.Start) > "" Then
        If CStr(attr).Length > 0 Then
            attr &= ","
        End If
        attr &= "start=" & laObject(FieldOrder.Start)
    End If

    'obs
    If CStr(attr).Length > 0 Then
        attr &= ","
    End If
    If laObject(FieldOrder.Data) > "" Then
        attr &= "obs=" & Split(laObject(FieldOrder.Data), ",").Count.ToString
    Else
        attr &= "obs=0"
    End If
End Sub
```

The attributes are returned to EViews in a single string containing a comma separated list of attributes. See Appendix A for a discussion of this and other formats that could have been used.

We also increment the `mLineIndex` pointer so that we're ready for the next call to `SearchNext` and then return `True` to indicate that we have a result. If we encounter the end of the file, we return `False` to indicate to EViews that the search is complete.

Interim Build Check

At this point, we are ready to test our new Custom Text Database Extension with EViews. First, build the project, checking that all registration steps completed successfully. (Failures to register objects will typically be caused by insufficient user permissions. You can resolve this by running the development environment as an administrator or by registering the objects yourself outside the development environment using an administrator account). Once the build completes without errors, launch a copy of EViews and register our new Database Extension by typing the following into the EViews command window:

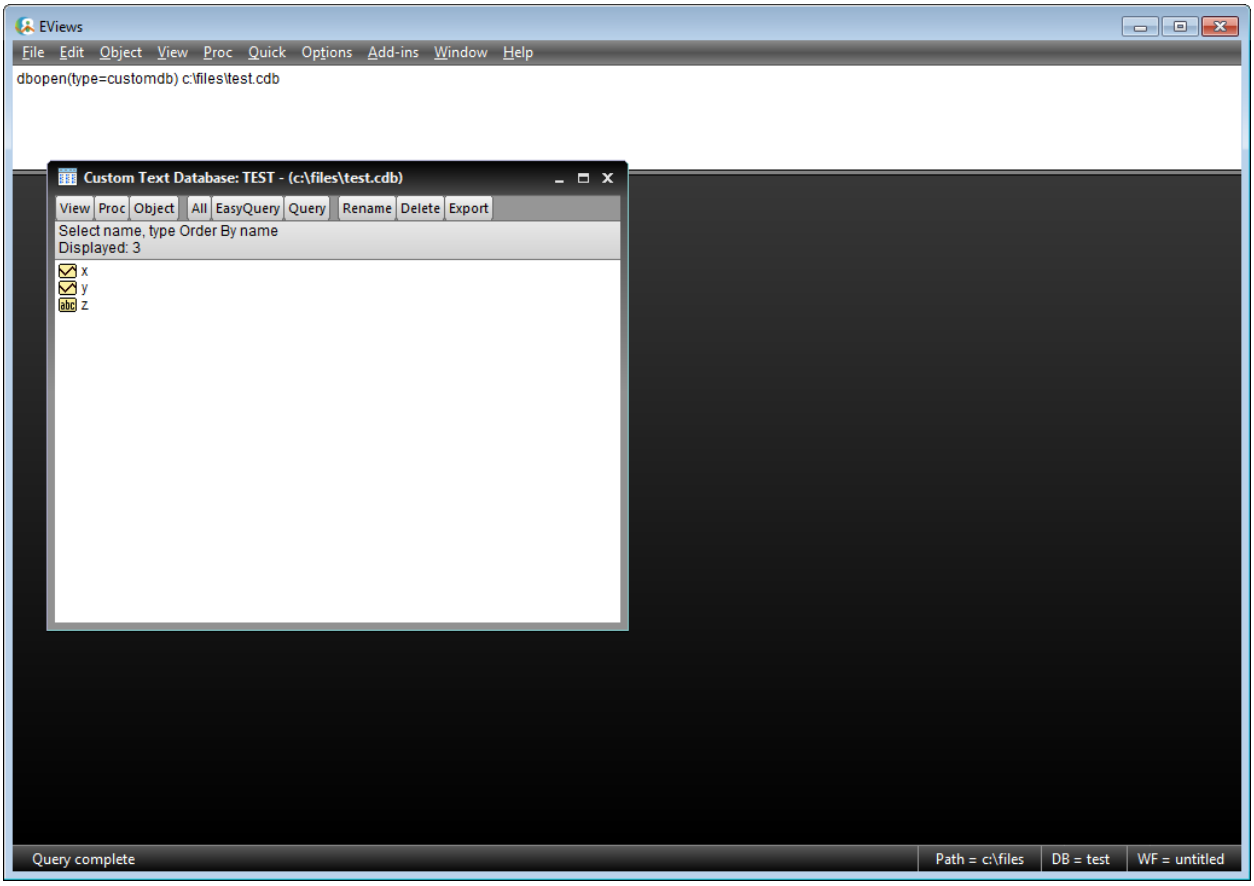
```
edxadd EdxSamples.CustomDbMgr
```

This registers our new Database Manager object with EViews (by supplying its ProgId) and makes it available for use. (Note that there is a matching command `edxdrop` that can be used to unregister a database extension.)

To test our new database format, we simply tell EViews to open our test file as a database:

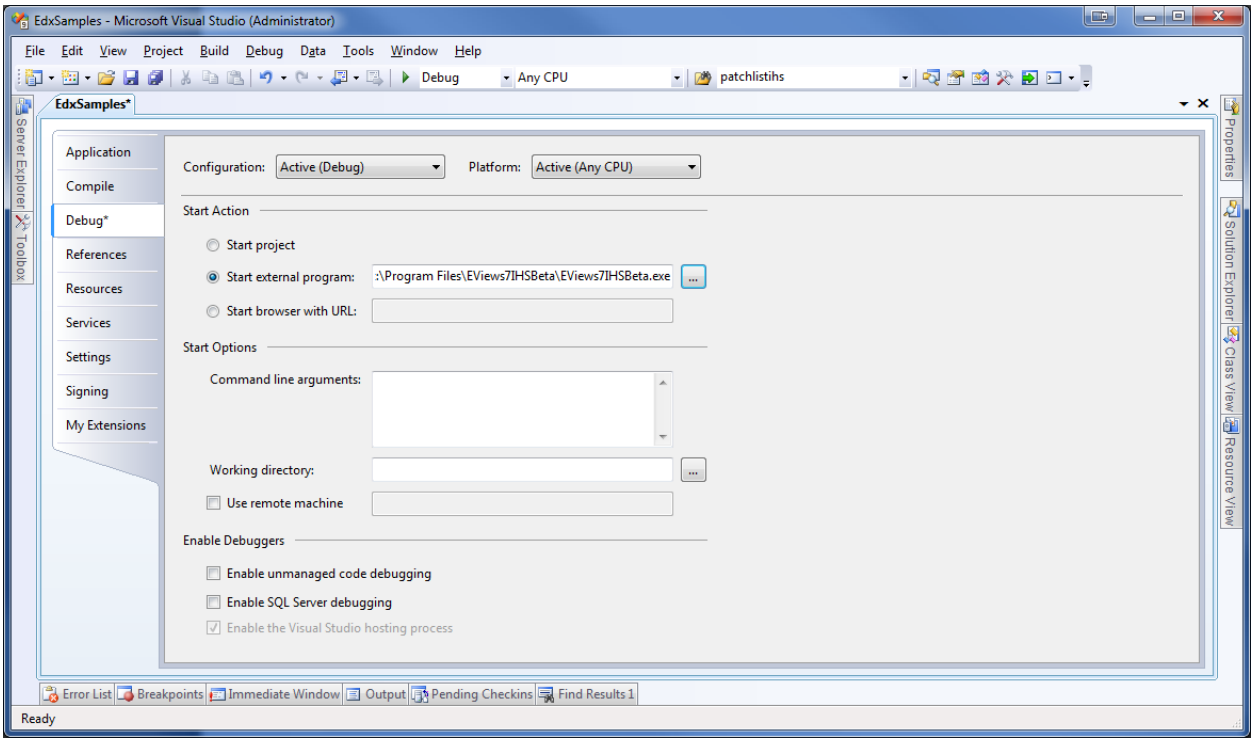
```
dbopen c:\files\test.cdb
```

You should see an empty database window. Click the All button to display the three items that are in our custom text file:



Debugging CustomEDX

You may find it extremely useful to setup your debugging environment in Visual Studio to run EViews every time you press the Debug button in Visual Studio. Right-click the "EdxSamples" project and click on "Properties". Click on the "Debug" tab on the left and select the "Start external program" radio button and type in the full path to your EViews7:



You can now place breakpoints in the methods of CustomDbMgr and CustomDb to see when EViews calls the functions and to examine what values are being passed in and out of the functions.

In order to support exporting of an object in our database to an EViews database or workfile, we need to add code to `ReadObjectAttributes` and `ReadObject`:

ReadObjectAttributes

`ReadObjectAttributes` is used by EViews to quickly get attribute data for an object without having to read any data values.

In our example, this doesn't save much in terms of performance (because our file is small and all the data in our file has already been loaded into memory), but more complicated databases may benefit from this separation. For now, we'll just throw the NotImplementedException which will cause EViews to call `ReadObject` instead.

```
Public Sub ReadObjectAttributes(ByVal objectId As String, _
                                ByVal destFreqInfo As String, _
                                ByRef attr As Object) _
        Implements EViewsEdx.IDatabase.ReadObjectAttributes
    Throw New NotImplementedException
End Sub
```

ReadObject

`ReadObject` is expected to retrieve all attributes and data values for the specified object:

```
Public Sub ReadObject(ByVal objectId As String, _
                      ByVal destFreqInfo As String, _
                      ByRef attr As Object, _
                      ByRef vals As Object, _
                      ByRef ids As Object) _
        Implements EViewsEdx.IDatabase.ReadObject
    For i As Integer = LBound(maLines) To UBound(maLines)
        If maLines(i).Length > 0 Then
            Dim laObject() As String = Split(maLines(i), "|")
            If UCase(laObject(FieldOrder.Name)) = UCase(objectId) Then
                'found it
                BuildAttributeString(maLines(i), attr)
                'get the data vals...
                vals = Split(laObject(FieldOrder.Data), ",")
                Return
             End If
        End If
    Next
    'the object doesn't exist in our database
    Throw New COMException("", EViewsEdx.ErrorCode.RECORD_NAME_INVALID)
End Sub
```

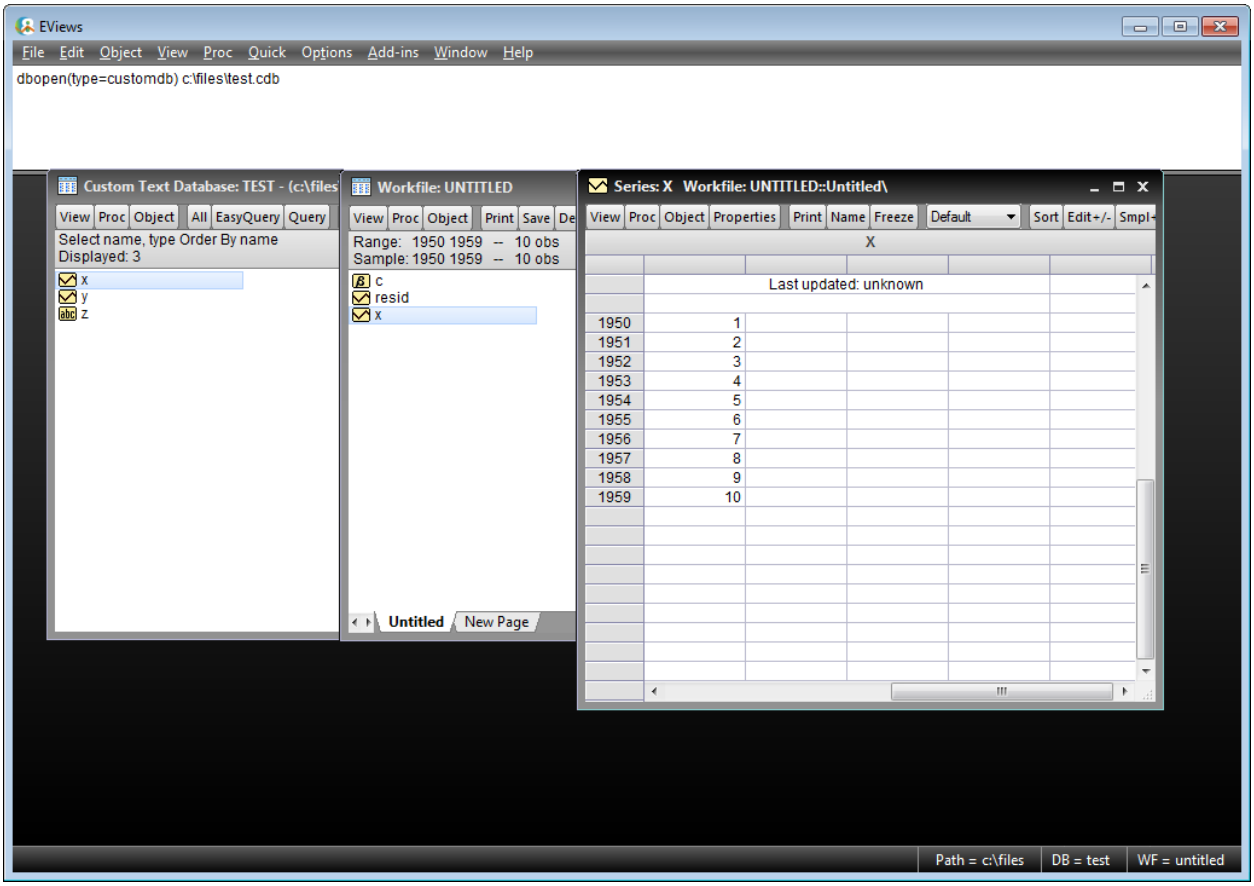
This method simply searches for the line containing the name in `objectId`, then builds the attribute string and parses out the data values into the `vals` array. If we cannot find the object name, we throw an exception to tell EViews why we failed.

Testing ReadObject

We can now test the Read methods by attempting to export an object from our database into a new workfile. Run EViews and open our database:

```
dbopen(type=customdb) c:\files\test.cdb
```

Click the "All" button to display all available objects in the database. Right-click the icon for the series X and select "Export to workfile…" and click "OK" on the Database Export dialog.



You should be able to confirm that the series X has been created in the new workfile and that it contains the 10 observations that were defined in the test.cdb text file.

Summary

We now have a completed Custom Database Extension that supports simple read-only access to data stored in a text file. We will now proceed to our second example that extends this to support writing to the database and also includes user configurable database preferences.

XML Folder Based Database

Our second example will be another file-based database, but each object will be stored in a separate XML file. The folder containing all of these XML files will represent the new database. Each XML file will contain both the data and the attributes for a single object. To simplify the XML generation and parsing, this example will use the standard System.Data.Dataset class available in the .NET Framework.

The complete source code for this example is provided in the EdxSamples project available at http://www.eviews.com/EViews8/Enterprise/EDXeg.html. You can examine and modify this to suit your own needs. You may find it useful to follow the steps below as you read through the discussion so you can better understand how the example project was created.

Create the XML folder

In the "EdxSamples" project, we will create a new folder named "XML" to group these files together. Right-click the "EdxSamples" project, then select "Add", then "New Folder". Name the folder "XML".

Create the Database Manager class

Right click the "XML" folder and select "Add", then "Class" from the menu. In the Add New Item dialog, select the "Class" template and rename the file to "XmlDbMgr.vb", then click Add.

As in the previous example, we need to tell Visual Studio to make this class visible to COM clients and specify its GUID value. We'll also tell Visual Studio not to generate a custom interface definition for this class as it is not needed. To do this, copy and paste the following lines in the XmlDbMgr.vb file above the "Public Class" line:

```
Imports System.Runtime.InteropServices

<Guid("XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX"), _
 ClassInterface(ClassInterfaceType.None), _
 ComVisible(True)> _
Public Class XmlDbMgr
```

This class will need to implement the EViewsEdx.IDatabaseManager interface. Below the "Public Class" line, add the following:

```
Public Class XmlDbMgr
    Implements EViewsEdx.IDatabaseManager
```

After adding this line, .NET will automatically add empty versions of each function that is part of our IDatabaseManager interface.

The first method we'll code is the GetAttributes method.

GetAttributes

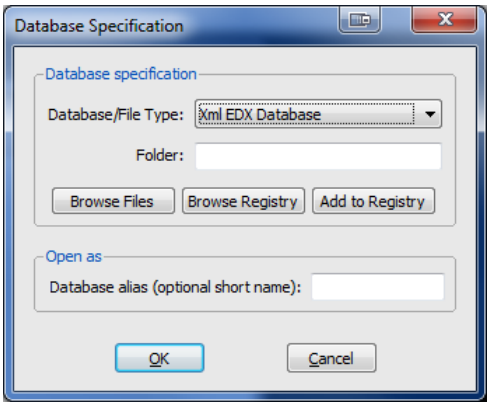
Following the same pattern as in the previous example, we return a list of important attributes about the database in a single comma delimited string.

```
Public Function GetAttributes(ByVal clientInfo As String) As Object _
        Implements EViewsEdx.IDatabaseManager.GetAttributes

 Dim lsAtts As String = "name=XmlEDX, description=Xml EDX Database, " & _
                        "type=xmledx, search=all|attr, searchattr=name, " & _
                        "attrtype=strarray, dbidlabel=Folder"

    Return lsAtts
End Function
```

Most of the attributes are the same as for the previous example, but we no longer specify a file extension since our database identifier will now be a directory. Note: One important side effect of not having an extension attribute is that when a user opens an instance of our XML database, it will not appear in the File menu's Most Recently Used (MRU) listing. Only file-based databases that have a file extension and those defined in the EViews Database registry (with a short name) will appear in the MRU listing. We have also removed the nocreate and readonly attributes since we will also support writing to the database in this example.

The `attrtype=strarray` attribute tells EViews to send us object attributes as a string array. Since we will store these attributes into a DataTable, this will make parsing the attributes much easier in the `WriteObject` method.

The `dbidlabel` attribute tells EViews to use a custom value (in our case "Folder") as the label next to the Database ID field in the Database Open and Database Create dialogs. This will help users know what to put into the field when performing a `dbopen` or `dbcreate`.



Please refer to Appendix B for details on Database Manager attributes.

Now that we've told EViews the general details of our database format, our manager needs to be able to return a specific database to EViews.

OpenDb

Whenever an EViews user opens or creates a database in our format, EViews calls the OpenDb method on IDatabaseManager to retrieve an IDatabase interface. The IDatabase interface represents a "connection" to our database and will be used by EViews to search our database and to read and write objects to our database.

```
Public Function OpenDb(ByVal databaseId As String, _
                       ByVal oc_mode As EViewsEdx.OpenCreateMode, _
                       ByVal rw_mode As EViewsEdx.ReadWriteMode, _
                       ByVal server As String, _
                       ByVal username As String, _
                       ByVal password As String) As EViewsEdx.IDatabase _
        Implements EViewsEdx.IDatabaseManager.OpenDb
```

```
    Return New XmlDb(databaseId, oc_mode, rw_mode)
End Function
```

Before we can use this method, we will need to define the XmlDb class that will be returned by this method.

Create the Database class

Like we did before, right click the "XML" folder in the Solution Explorer and select "Add", then "Class" from the menu. In the Add New Item dialog, select the "Class" template and rename the file to "XmlDb.vb", then click Add.

As before, this class does not need to be ComVisible as EViews will never instantiate it directly.

XmlDb will need to implement the EViewsEdx.IDatabase interface in order for it to be usable by EViews:

```
Imports System.Runtime.InteropServices

Public Class XmlDb
    Implements EViewsEdx.IDatabase
```

After adding this line, .NET will automatically add empty versions of each function that is required by our IDatabase interface. We'll begin by adding some class level variables:

```
'class level variables
Private msDatabaseId As String
Private mOpenCreateMode As EViewsEdx.OpenCreateMode
Private mReadWriteMode As EViewsEdx.ReadWriteMode

Private mSearchExpression As String
Private mFiles() As String
Private mUpper As Integer
Private mIndex As Integer
```

Also add a new class constructor:

```
'new constructor
Public Sub New(ByVal databaseId As String, _
               ByVal oc_mode As EViewsEdx.OpenCreateMode, _
               ByVal rw_mode As EViewsEdx.ReadWriteMode)
    MyBase.New()

    msDatabaseId = databaseId
    mOpenCreateMode = oc_mode
    mReadWriteMode = rw_mode
    mUpper = 0
    mIndex = 0

    Dim lbDirExists As Boolean = System.IO.Directory.Exists(msDatabaseId)

    Select Case oc_mode
        Case EViewsEdx.OpenCreateMode.FileOpen
```

```
            If Not lbDirExists Then
                Throw New COMException(String.Empty, _
                    EViewsEdx.ErrorCode.FILE_FILENAME_INVALID)
            End If

        Case EViewsEdx.OpenCreateMode.FileCreate
            If lbDirExists Then
                Throw New COMException(String.Empty, _
                    EViewsEdx.ErrorCode.FILE_FILENAME_IN_USE)
            Else
                'create the new subdirectory...
                Util.CreateSubDirectory(msDatabaseId)
            End If

        Case EViewsEdx.OpenCreateMode.FileOverwrite
            If lbDirExists Then
                'delete the directory first...
                System.IO.Directory.Delete(msDatabaseId, True)
                lbDirExists = False
            End If
            'create the new subdirectory...
            Util.CreateSubDirectory(msDatabaseId)

        Case EViewsEdx.OpenCreateMode.FileOpenCreate
            'create if not already existing
            If Not lbDirExists Then
                Util.CreateSubDirectory(msDatabaseId)
            End If
    End Select
End Sub
```

First, we save all passed in parameter values into member variables so we can refer to them later: `databaseId` will contain the path to the folder, `oc_mode` and `rw_mode` will tell us how our database was opened.

Since this database supports creation, our code may need to make a new folder or delete an existing folder depending on the `oc_mode` passed in.

When we encounter an error, we throw a `COMException` object that uses a pre-defined EViews error constant so that EViews will know how to respond to that error. For example, if `oc_mode` is `FileOpen` and the specified folder doesn't exist, we throw a `FILE_FILENAME_INVALID COMException` which will instruct EViews to display the "Database not found" error message. The API documentation for the `OpenDb` function contains a list of exceptions that may be relevant.

For this example, we don't bother looking for any object files in the specified folder until a Search request is made by the user.

Util Class

You may have noticed that we make use of a class named "Util" that contains some global utility functions. Create a new Util class somewhere in your project and place the following code into that class:

```
Imports System.IO

Public Class Util
    Public Shared Function myCInt(ByRef roValue As Object) As Integer
        Try
            If roValue Is DBNull.Value Then
                Return 0
            End If
            Return CInt(roValue)
        Catch ex As Exception
            Return 0
        End Try
    End Function

    Public Shared Sub CreateSubDirectory(ByVal vsPath As String)
        Dim di As New System.IO.DirectoryInfo(vsPath)
        If di.Exists Then
            Return
        End If

        di.Create()
    End Sub
End Class
```

SearchByAttributes

```
Public Sub SearchByAttributes(ByVal searchExpression As String, _
                              ByVal attrNames As String) _
        Implements EViewsEdx.IDatabase.SearchByAttributes

    'store the search expression
    mSearchExpression = searchExpression

    mFiles = System.IO.Directory.GetFiles( _
                msDatabaseId, _
                mSearchExpression & ".xml")

    'reset any previous search pointer...
    mUpper = UBound(mFiles)
    mIndex = 0
End Sub
```

Remember that EViews will first call `SearchByAttributes` to allow the database to prepare a list of database objects to return. EViews will then call `SearchNext` repeatedly to retrieve the name and attributes of each object in the list until it has retrieved the full list.

For this example, searching the database will involve iterating over the XML files contained in the directory specified by the `databaseId`.

Since we included `SEARCHATTR=name` in `GetAttributes`, the `searchExpression` argument will contain a string that represents a name pattern (e.g. "*" or "gdp*"). We will use this expression in our call to `Directory.GetFiles` to return only those objects whose name fits this pattern. We store the results of this search in our `mFiles` variable.

We also reset the `mIndex` value so that the first call to `SearchNext` after calling this function will always start at the beginning of the directory listing.

SearchNext

```
Public Function SearchNext(ByRef objectId As String, _
                           ByRef attr As Object) As Boolean _
        Implements EViewsEdx.IDatabase.SearchNext

TryAgain:
    If mIndex > mUpper Then
        Return False
    End If

    Dim temp As String = mFiles(mIndex)
    mIndex += 1

    Dim fi As New System.IO.FileInfo(temp)
    temp = fi.Name
    Dim pos As Integer = InStrRev(temp, ".")
    If pos > 0 Then
        temp = Mid(temp, 1, pos - 1)
    End If

    objectId = temp

    'try to get the attributes...
    Dim liSecondDimSize As Integer
    Try
        Dim ds As New DataSet
        Dim dtAttributes As DataTable
        Dim dtMeta As DataTable
        Dim dtData As DataTable

        ds.ReadXml(fi.FullName)
        dtAttributes = ds.Tables("Attributes")
        dtMeta = ds.Tables("Meta")
        dtData = ds.Tables("Data")

        If dtAttributes Is Nothing Or dtMeta Is Nothing _
        Or dtData Is Nothing Then
            GoTo TryAgain
        End If

        If dtAttributes.Rows.Count > 0 Then
            attr = GetAttributesAsObject(dtAttributes)
        End If

        'verify that we have a meta second dim size value as well
```

```
        If dtMeta.Rows.Count > 0 Then
          liSecondDimSize = Util.myCInt(dtMeta.Rows(0).Item("SecondDimSize"))
        End If

        Return True

    Catch ex As Exception
        'we encountered a file that wasn't a valid dataset xml
        'skip it and try the next one...
        GoTo TryAgain
    End Try
End Function
```

Every time `SearchNext` is called, it retrieves the next object file in `mFiles`.

In our example, the object file is an XML representation of an object in the database. To verify this, we attempt to load it into a new Dataset object. Once loaded, we retrieve any object attributes that were previously saved using a helper method named `GetAttributesAsObject`, and return that in `attr`.

```
Private Function GetAttributesAsObject(ByRef rdtAttributes As DataTable) _
        As Object
    Dim liColCount As Integer = rdtAttributes.Columns.Count
    Dim loArray(0 To (liColCount - 1), 0 To 1) As String

    For i As Integer = 0 To (liColCount - 1)
        loArray(i, 0) = rdtAttributes.Columns(i).ColumnName
        loArray(i, 1) = rdtAttributes.Rows(0).Item(loArray(i, 0))
    Next

    Return loArray
End Function
```

If a valid object is found, the function returns TRUE. Once we reach the end of `mFiles`, we return FALSE to tell EViews that we are done.

Interim Build Check

At this point, we can test our new Database Extension with EViews. Build the project, checking that all registration steps completed successfully. Once the build completes without errors, launch a copy of EViews and register our new Database Extension by typing the following into the EViews command window:

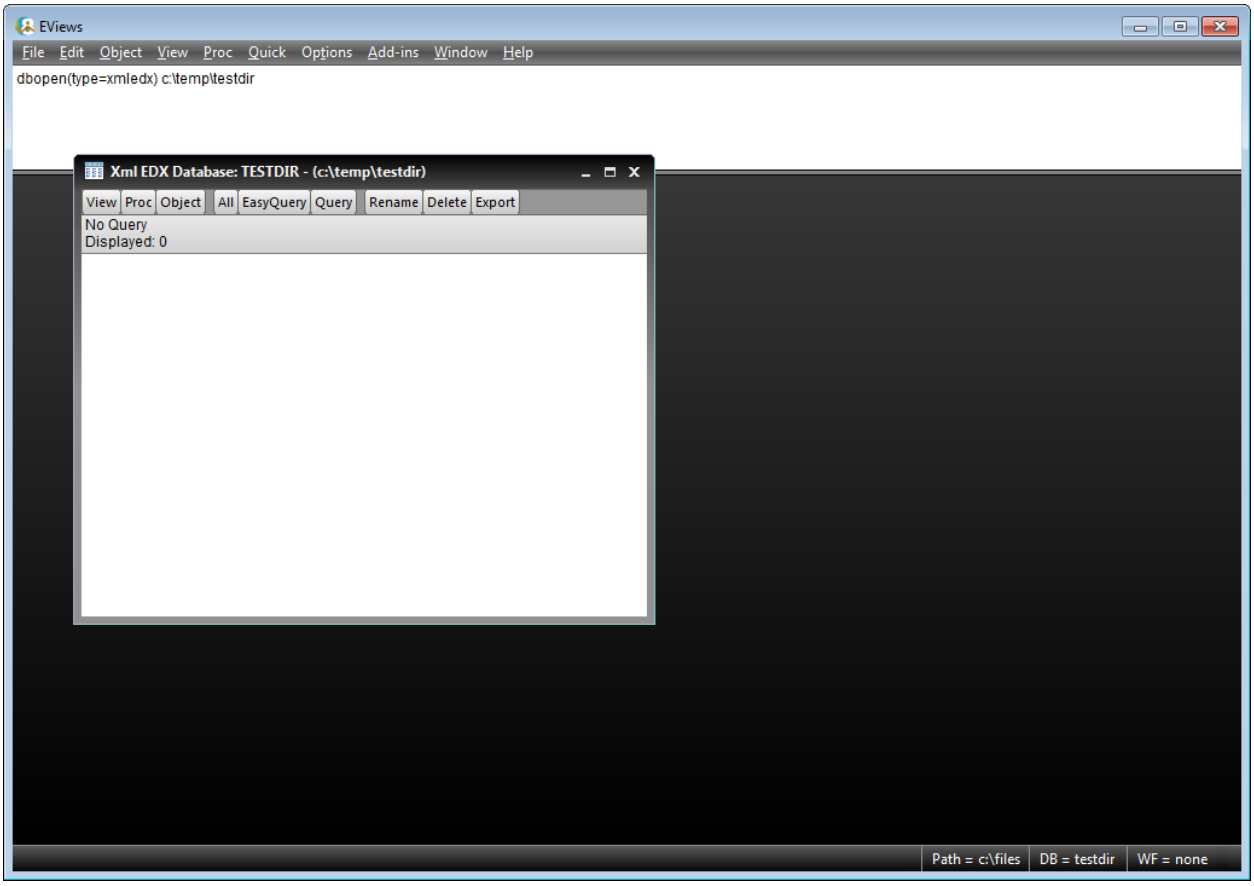
```
edxadd EdxSamples.XmlDbMgr
```

This registers our new Database Manager object with EViews and makes it available for use.

To test our new database format, first create an empty subdirectory somewhere on your computer (e.g. c:\temp\TestDir). Then in EViews, call

```
dbopen(type=xmledx) c:\temp\testdir
```

Note that we need to use the 'type=' option since our database consists of an entire directory so it does not have a simple file extension. You should see an empty database window.



If you click the All button, you should see zero objects returned since our directory is initially empty.

You will notice that because we did not define an extension attribute for our database (see `GetAttributes`), this database does not appear under the File menu's Most Recently Used (MRU) file listing. This is because without a file extension, EViews does not have a way to determine the database type from the path alone. Furthermore, all server-based database extensions (such as our SQL example below) do not appear in the MRU listing as well. The only way for these databases to show up in the listing is to be pre-defined in the EViews Database Registry with a short name (which can be done during the DBOPEN dialog).

WriteObject

We need some test objects to read from our database. The quickest way to do this is to allow EViews to write objects into our database. Supporting write simply means we have to code the IDatabase::WriteObject method.

First, we'll write a helper function to determine the size of the object arrays that are passed into WriteObject:

```
Public Sub DetermineSize(ByRef roObj As Object, _
                         ByRef riFirst As Integer, _
                         ByRef riSecond As Integer)
    If roObj Is Nothing Then
        riFirst = 0
        riSecond = 0
        Return
    End If

    Try
        riFirst = UBound(roObj, 1) - LBound(roObj, 1) + 1
    Catch ex As Exception
        riFirst = 0
    End Try

    Try
        riSecond = UBound(roObj, 2) - LBound(roObj, 2) + 1
    Catch ex As Exception
        riSecond = 0
    End Try
End Sub
```

Now, WriteObject:

```
Public Sub WriteObject(ByRef objectId As String, _
                       ByVal attr As Object, _
                       ByVal vals As Object, _
                       ByVal ids As Object, _
                       ByVal overwriteMode As EViewsEdx.WriteType) _
        Implements EViewsEdx.IDatabase.WriteObject

    Dim lsFilePath As String = msDatabaseId & "\" & LCase(objectId) & ".xml"

    Select Case overwriteMode
        Case EViewsEdx.WriteType.WriteProtect
            'if the file already exists, don't overwrite it...
            If System.IO.File.Exists(lsFilePath) Then
                Throw New COMException("",
EViewsEdx.ErrorCode.RECORD_NAME_IN_USE)
            End If

        Case EViewsEdx.WriteType.WriteOverwrite
            If System.IO.File.Exists(lsFilePath) Then
                System.IO.File.Delete(lsFilePath)
            End If
    End Select

    Dim ds As New DataSet
    Dim dt As New DataTable

    'save the attributes
    dt.TableName = "Attributes"
    For i As Integer = LBound(attr) To UBound(attr)
        dt.Columns.Add(attr(i, 0))
    Next
```

```
    Dim dr As DataRow = dt.NewRow
    For i As Integer = LBound(attr) To UBound(attr)
        dr(attr(i, 0)) = attr(i, 1)
    Next
    dt.Rows.Add(dr)
    ds.Tables.Add(dt)

    'save our meta data
    Dim liFirstDim As Integer
    Dim liSecondDim As Integer
    DetermineSize(vals, liFirstDim, liSecondDim)

    dt = New DataTable("Meta")
    dt.Columns.Add("SecondDimSize")
    dr = dt.NewRow
    dr("SecondDimSize") = liSecondDim
    dt.Rows.Add(dr)

    ds.Tables.Add(dt)

    dt = New DataTable("Data")
    If ids IsNot Nothing Then
        dt.Columns.Add("id")
    End If
    If liSecondDim > 0 Then
        For y As Integer = 1 To liSecondDim
            dt.Columns.Add("value" & y.ToString)
        Next
    Else
        dt.Columns.Add("value")
    End If

    Dim lowerbound As Integer = LBound(vals)
    Dim upperbound As Integer = UBound(vals)

    For i As Integer = lowerbound To upperbound
        dr = dt.NewRow
        If ids IsNot Nothing Then
            dr("id") = ids(i)
        End If
        If liSecondDim = 0 Then
            dr("value") = vals(i)
        Else
            For y As Integer = 1 To liSecondDim
                dr("value" & y.ToString) = vals(i, y - 1)
            Next
        End If
        dt.Rows.Add(dr)
    Next
    ds.Tables.Add(dt)

    'now save this dataset as xml...
    ds.WriteXml(lsFilePath)
End Sub
```

Our `WriteObject` method will store all object attributes, values, and ids (if available) into a single Dataset object that contains three DataTables. The first DataTable is named "Attributes" and will contain a column for each attribute name. This table will only contain a single row that stores the attribute values for each name (which is returned by EViews as a string array because we specified `attrtype=strarray` in `GetAttributes`). The second DataTable is named "Meta" and will contain a single column named `SecondDimSize`. We store the size of the second dimension of the `vals` array (if it has one) so that when we read it back later, we know how many columns to read in (so that we can support matrix objects).

The third table is named "Data" and will contain a column named "id" (if `ids` were passed in) and then a value column for each column in the `vals` array. The number of rows in the DataTable will match the size of the `ids` and `vals` arrays that are passed in by EViews.

Once everything is placed into the new DataSet, we call `WriteXml` to save the dataset using the filename we generated from the passed in `objectId`.

Testing WriteObject

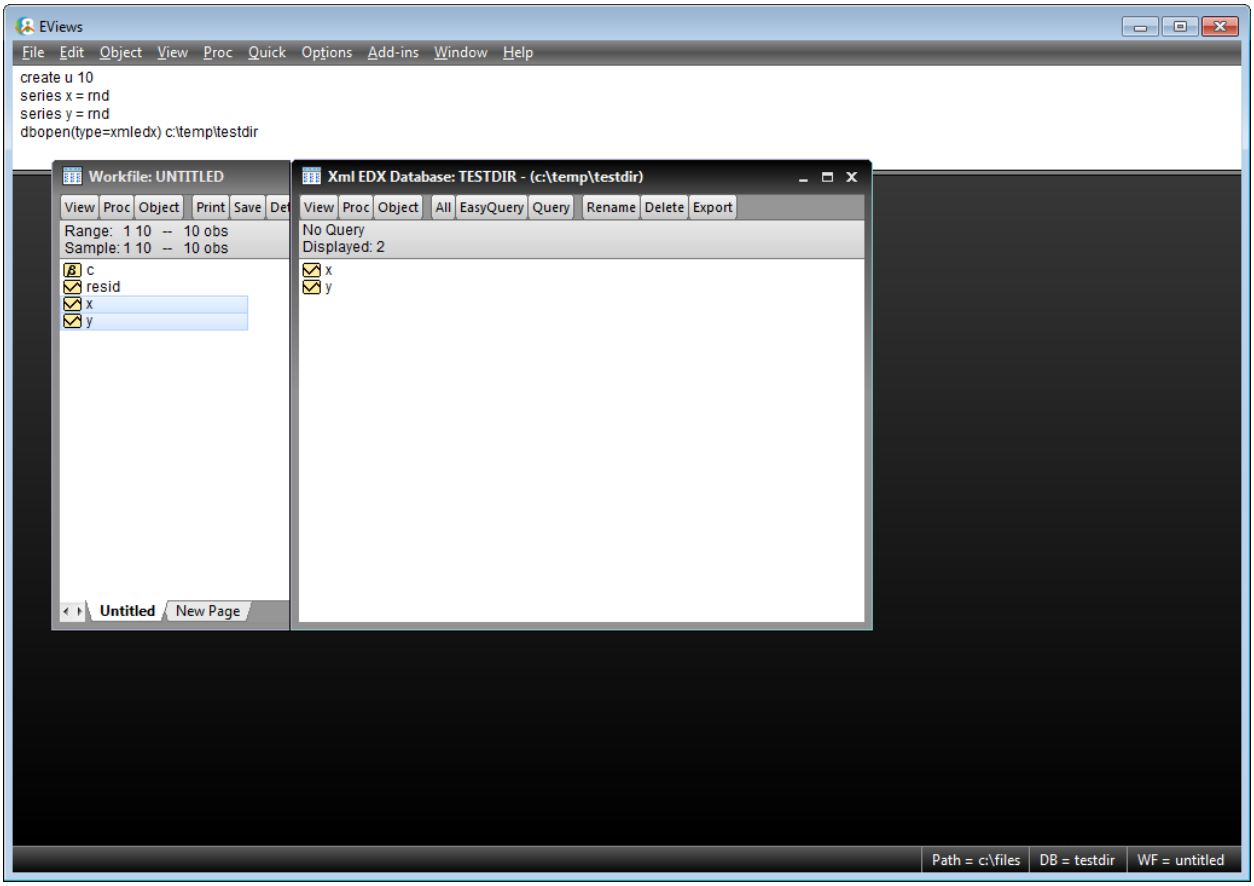
Now that we've completed the `WriteObject` method, we can run EViews and put various objects into our new database. Run EViews and run the following commands:

```
create u 10
series x = rnd
series y = rnd
```

Next, call dbopen again to open our database:

```
dbopen(type=xmledx) c:\temp\TestDir
```

Now drag and drop series X and Y onto our database.



Confirm that you have two new XML files in the database folder.

Now that we have objects in our database, we can code the `ReadObjectAttributes` and `ReadObject` methods to support reading.

ReadObjectAttributes

`ReadObjectAttributes` is used by EViews to quickly get attribute data for an object without having to read in any data values. As for our previous example, this doesn't save much in terms of performance here, so we'll just throw the NotImplementedException which will cause EViews to call ReadObject for the attributes instead.

```
Public Sub ReadObjectAttributes(ByVal objectId As String, _
                                ByVal defaultFreq As String, _
                                ByRef attr As Object) _
        Implements EViewsEdx.IDatabase.ReadObjectAttributes
    Throw New NotImplementedException()
End Sub
```

ReadObject

`ReadObject` is called to retrieve all attributes, data values and observation ids for the specified object:

```
Public Sub ReadObject(ByVal objectId As String, _
                      ByVal defaultFreq As String, _
                      ByRef attr As Object, _
                      ByRef vals As Object, _
                      ByRef ids As Object) _
        Implements EViewsEdx.IDatabase.ReadObject
    Dim lsFilePath As String = msDatabaseId & "\" & _
                               LCase(objectId) & ".xml"

    If Not System.IO.File.Exists(lsFilePath) Then
        Throw New COMException("", EViewsEdx.ErrorCode.RECORD_NAME_INVALID)
    End If

    Dim ds As New DataSet
    Dim dtAttributes As DataTable
    Dim dtMeta As DataTable
    Dim dtData As DataTable
    Dim liSecondDimSize As Integer = 0

    Try
        ds.ReadXml(lsFilePath)
        dtAttributes = ds.Tables("Attributes")
        dtMeta = ds.Tables("Meta")
        dtData = ds.Tables("Data")

        If dtAttributes.Rows.Count > 0 Then
            attr = GetAttributesAsObject(dtAttributes)
        End If

        If dtMeta.Rows.Count > 0 Then
          liSecondDimSize = Util.myCInt(dtMeta.Rows(0).Item("SecondDimSize"))
        End If

        If liSecondDimSize = 0 Then
           ReDim vals(0 To dtData.Rows.Count - 1)
        Else
           ReDim vals(0 To dtData.Rows.Count - 1, 0 To (liSecondDimSize - 1))
        End If

        If dtData.Columns.Contains("ids") Then
            ReDim ids(0 To dtData.Rows.Count - 1)
        End If

        Dim i As Integer = 0
        For Each dr As DataRow In dtData.Rows
            If liSecondDimSize = 0 Then
                vals(i) = dr("value")
            Else
                For y As Integer = 1 To liSecondDimSize
                    vals(i, y - 1) = dr("value" & y.ToString)
                Next
            End If
            If dtData.Columns.Contains("ids") Then
                ids(i) = dr("id")
            End If
            i += 1
        Next
```

```
    Catch ex As Exception
        Throw New COMException("Specified xml file was not valid.")
    End Try
End Sub
```

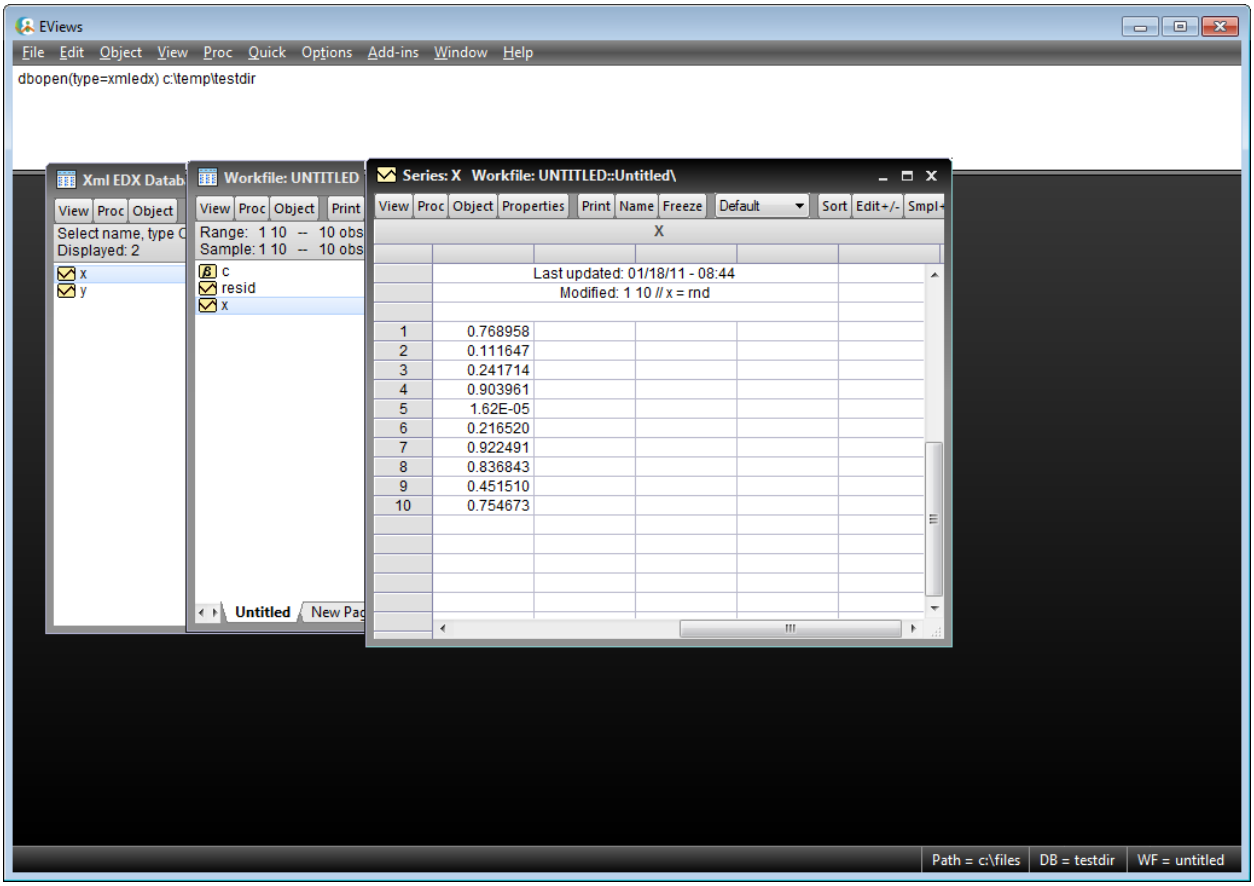
This method simply loads the specific object XML file and returns the attributes stored in the "Attributes" DataTable into the `attr` parameter. It then sizes and fills the `vals` and `ids` parameters with the associated data from the "Data" DataTable.

Testing ReadObject

We can now test the Read methods by attempting to export the X and Y objects from our database into a new workfile. Run EViews and open our database:

```
dbopen(type=xmledx) c:\temp\TestDir
```

Click the "All" button to display all available objects in the database. Right-click the X series object and select "Export to workfile…" and click "OK" on the Database Export dialog.

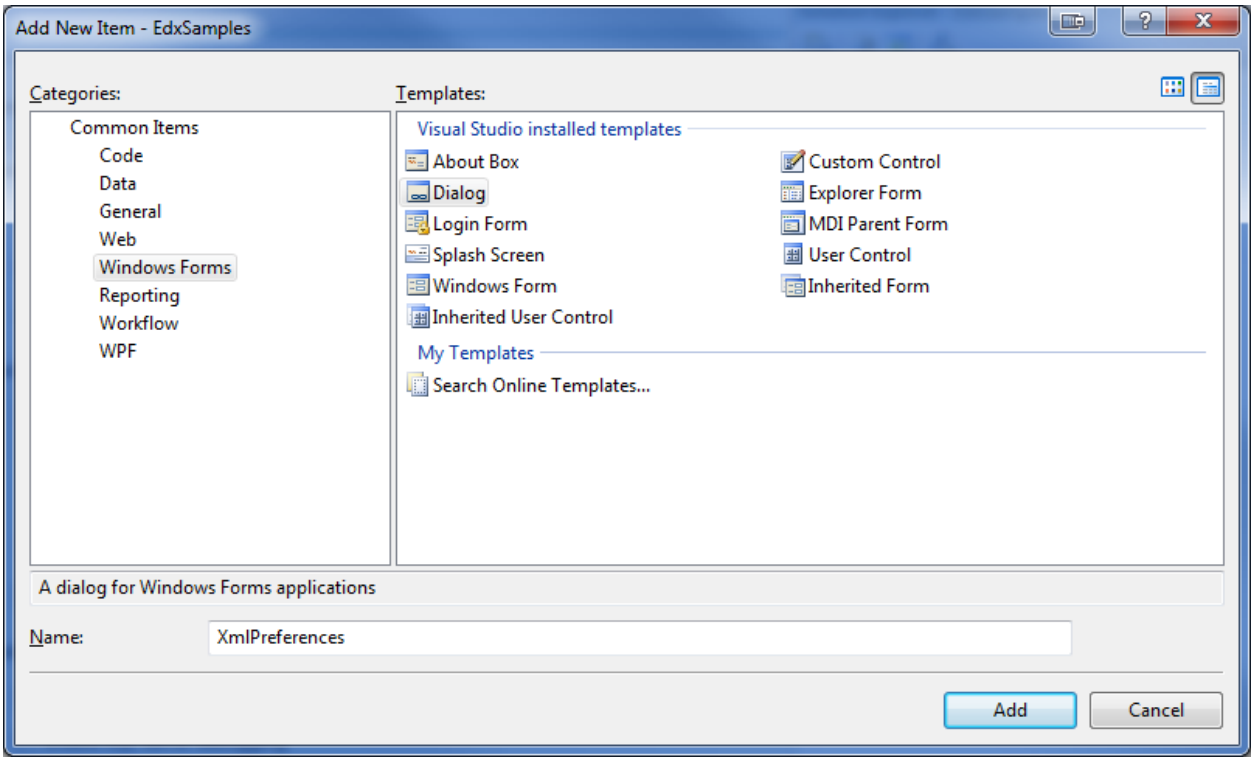


You should be able to confirm that the X series object is created in the new workfile correctly and that it contains the 10 observations that were generated randomly during the Write test.

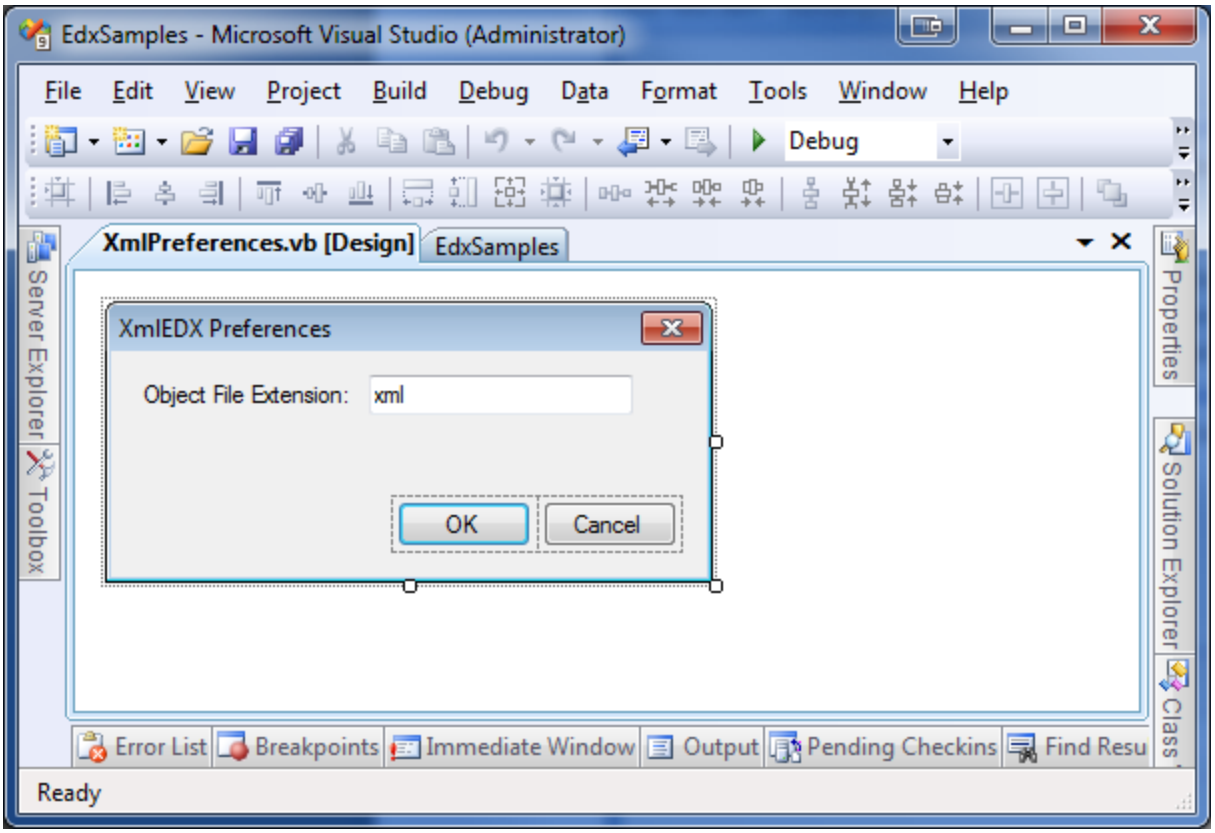
Supporting User Configurable Preferences

We will now add support for a database preference that is configurable by the user. In our example, we will let the user change the name of the extension for the object files. The default value will be ".xml" but they can change it to any other value.

First, we will need a dialog that the user can interact with to view and edit this value. Right-click the "XML" folder and click Add->Windows Form…. Select Windows Forms in the left tree view and then select Dialog on the right. Name the form XmlPreferences.vb, and click Add:



Add a label and a textbox (named `txtObjectExt`) control:



The code behind for this form should look like this:

```
Imports System.Windows.Forms

Public Class XmlPreferences

    Private Sub OK_Button_Click(ByVal sender As System.Object, _
                                ByVal e As System.EventArgs) _
                                Handles OK_Button.Click
        txtObjectExt.Text = Trim(txtObjectExt.Text)
        If txtObjectExt.Text.StartsWith(".") Then
            txtObjectExt.Text = Mid(txtObjectExt.Text, 2)
        End If
        If txtObjectExt.Text = "" Then
            MsgBox("You must specify a non-blank object file extension
value.", MsgBoxStyle.Critical)
            Return
        End If

        Me.DialogResult = System.Windows.Forms.DialogResult.OK
        Me.Close()
    End Sub

    Private Sub Cancel_Button_Click(ByVal sender As System.Object, _
                                    ByVal e As System.EventArgs) _
                                    Handles Cancel_Button.Click
```

```
        Me.DialogResult = System.Windows.Forms.DialogResult.Cancel
        Me.Close()
    End Sub

End Class
```

To use this form, we'll add code to the `ConfigurePreferences` method in the XmlDbMgr class. But before we do that, we'll need a class to store our preferences. In this case we're only storing a single string so this class will be minimal. Create a new class named XmlPrefs:

```
Public Class XmlPrefs
    Private msObjectFileExt As String

    Public Sub New()
        msObjectFileExt = "xml"
    End Sub

    Public Property ObjectFileExt() As String
        Get
            Return msObjectFileExt
        End Get
        Set(ByVal value As String)
            msObjectFileExt = value
        End Set
    End Property
End Class
```

We'll create a member variable in our XmlDbMgr class that uses this class:

```
Public Class XmlDbMgr
    Implements EViewsEdx.IDatabaseManager

    Private mDbPref As New XmlPrefs
```

ConfigurePreferences

We'll code `ConfigurePreferences` to display the form, then save any changed values back into our XmlPrefs class and also into the `prefs` parameter:

```
Public Function ConfigurePreferences(ByVal server As String, _
                                     ByVal username As String, _
                                     ByVal password As String, _
                                     ByRef prefs As String) As Boolean _
        Implements EViewsEdx.IDatabaseManager.ConfigurePreferences

    Dim frm As New XmlPreferences
    frm.txtObjectExt.Text = LCase(mDbPref.ObjectFileExt)

    Dim result As System.Windows.Forms.DialogResult = frm.ShowDialog()
    If result = Windows.Forms.DialogResult.OK Then
        mDbPref.ObjectFileExt = LCase(frm.txtObjectExt.Text)
```

```
        prefs = "ObjectFileExt=" & mDbPref.ObjectFileExt
        Return True
    End If

    Return False
End Function
```

Returning `True` from this method will instruct EViews to save the new `prefs` value into the EViews INI file of the current user. The next time this Database Manager is loaded, it will be initialized with this `prefs` value by a call to `SetPreferences`:

SetPreferences

```
Public Sub SetPreferences(ByVal prefs As String) _
        Implements EViewsEdx.IDatabaseManager.SetPreferences

    Dim atts() As String = Split(prefs, ",")
    Dim nm As String
    Dim val As String
    Dim pos As Integer
    For Each att As String In atts
        If att > "" Then
            pos = InStr(att, "=")
            If pos > 0 Then
                nm = Mid(att, 1, pos - 1)
                val = Mid(att, pos + 1)
            Else
                nm = att
                val = att
            End If

            Select Case LCase(nm)
                Case "objectfileext"
                    mDbPref.ObjectFileExt = val
            End Select

        End If
    Next
End Sub
```

The only thing left to do now is to use this new user configurable option in our code to read and write objects in our XmlDb class. To do this, we'll change our XmlDb constructor to receive a reference to the `mDbPref` member variable maintained by XmlDbMgr so that XmlDb can read the current values. To keep the reference, we'll need another `mDbPref` member variable in XmlDb:

```
Private mDbPref As XmlPrefs
```

Constructor changes:

```
Public Sub New(ByVal databaseId As String, _
               ByVal oc_mode As EViewsEdx.OpenCreateMode, _
               ByVal rw_mode As EViewsEdx.ReadWriteMode, _
```

```
                ByRef roDbPref As XmlPrefs)
    MyBase.New()

    msDatabaseId = databaseId
    mOpenCreateMode = oc_mode
    mReadWriteMode = rw_mode
    mDbPref = roDbPref
```

Modify `OpenDb` to pass in `mDbPref`:

```
Public Function OpenDb(ByVal databaseId As String, _
                       ByVal oc_mode As EViewsEdx.OpenCreateMode, _
                       ByVal rw_mode As EViewsEdx.ReadWriteMode, _
                       ByVal server As String, ByVal username As String, _
                       ByVal password As String) As EViewsEdx.IDatabase _
                       Implements EViewsEdx.IDatabaseManager.OpenDb
    Return New XmlDb(databaseId, oc_mode, rw_mode, mDbPref)
End Function
```

Now, wherever we used a hard coded value for "`.xml`", we replace this with "`.`" `&` `mDbPref.ObjectFileExt`.

These changes should be made to `ReadObject`, `SearchByAttributes`, and `WriteObject`:

For example, in `SearchByAttributes`:

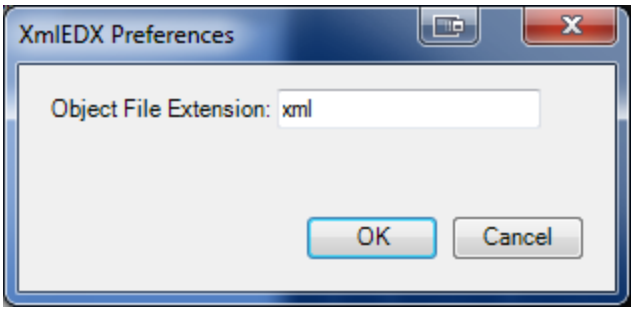
```
Public Sub SearchByAttributes(ByVal searchExpression As String, _
                              ByVal attrNames As String) _
        Implements EViewsEdx.IDatabase.SearchByAttributes
  'store the search expression
  mSearchExpression = searchExpression

  mFiles = System.IO.Directory.GetFiles( _
              msDatabaseId, _
              mSearchExpression & "." & mDbPref.ObjectFileExt)
```

Testing ConfigurePreferences

Once compiled, run EViews and open our database:

```
dbopen(type=xmledx) c:\temp\TestDir
```

Once opened, click the View button and select "Preferences…":



To make sure this Object Extension value is being saved properly, change it to something else (e.g. "evx") and click OK. Restart EViews and then go back to the dialog to see if it returns properly.

Changing the extension will not automatically refresh the display of objects in the Database window in EViews. You will have to hit the All button to do a refresh.

Now the only things left to implement for a fully functional database are the `DeleteObject`, `CopyObject`, and `RenameObject` methods.

DeleteObject

`DeleteObject` just does a simple file delete on the specified object:

```
Public Sub DeleteObject(ByVal objectId As String) _
        Implements EViewsEdx.IDatabase.DeleteObject

    Dim lsFilePath As String = msDatabaseId & "\" & _
                               LCase(objectId) & "." & _
                               mDbPref.ObjectFileExt
    If System.IO.File.Exists(lsFilePath) Then
        System.IO.File.Delete(lsFilePath)
    End If
End Sub
```

CopyObject

`CopyObject` just does a simple file copy on the specified object:

```
Public Sub CopyObject(ByVal srcObjectId As String, _
                      ByRef destObjectId As String, _
                      Optional ByVal overwrite As Boolean = False) _
        Implements EViewsEdx.IDatabase.CopyObject

    Dim lsSrcFilePath As String = msDatabaseId & "\" & _
                                  LCase(srcObjectId) & "." & _
                                  mDbPref.ObjectFileExt
    Dim lsDestFilePath As String = msDatabaseId & "\" & _
                                   LCase(destObjectId) & "." & _
                                   mDbPref.ObjectFileExt
    If System.IO.File.Exists(lsDestFilePath) And Not overwrite Then
        Throw New COMException("", EViewsEdx.ErrorCode.RECORD_NAME_IN_USE)
    End If
```

```
    Dim fi As New System.IO.FileInfo(lsSrcFilePath)
    fi.CopyTo(lsDestFilePath)
End Sub
```

The `srcObjectId` parameter is the object being copied. The `destObjectId` parameter is the name of the new object to copy to. If `overwrite` is `False` and the new object name already exists, we need to throw the `RECORD_NAME_IN_USE` error to notify EViews that the new object already exists. This will prompt EViews to display an Overwrite dialog to the user.

Actually, a simpler way to implement a copy operation would be to tell EViews that we didn't implement this function ourselves (by throwing the `NotImplementedException` exception like in `ReadObjectAttributes`). EViews would then try to do the copy manually by calling `ReadObject`, followed by `WriteObject`. This would have worked fine for us as well.

RenameObject

`RenameObject` does a simple file rename on the specified object:

```
Public Sub RenameObject(ByVal srcObjectId As String, _
                        ByVal destObjectId As String) _
        Implements EViewsEdx.IDatabase.RenameObject

    Dim lsSrcFilePath As String = msDatabaseId & "\" & _
                                  LCase(srcObjectId) & "." & _
                                  mDbPref.ObjectFileExt
    If Not System.IO.File.Exists(lsSrcFilePath) Then
        Throw New COMException("", EViewsEdx.ErrorCode.RECORD_NAME_INVALID)
    End If

    Dim lsDestFilePath As String = msDatabaseId & "\" & _
                                   LCase(destObjectId) & "." & _
                                   mDbPref.ObjectFileExt
    If System.IO.File.Exists(lsDestFilePath) Then
        Throw New COMException("", EViewsEdx.ErrorCode.RECORD_NAME_IN_USE)
    End If

    Dim fi As New System.IO.FileInfo(lsSrcFilePath)
    fi.MoveTo(lsDestFilePath)
End Sub
```

The `srcObjectId` parameter is the object being renamed. The `destObjectId` parameter is the new name. If the new name conflicts with any pre-existing object, we need to throw the `RECORD_NAME_IN_USE` error so EViews can show this error to the user.

Summary

We have now completed the XML Database Extension example. We will now proceed to our third example that shows how you might support a server-based database.

SQL Server Database

Our third example will be based on a SQL Server database. We will create an EViews Database Extension that will work for any SQL Server database. Because of the difficulties in knowing how to write data properly into an unknown SQL Server table, we'll keep this example Read Only.

Our server-based database extension will require the user to enter a server name, userid, and password. We will also require them to select a SQL Server Catalog in order to restrict the number of objects that will appear within a single database.

Once a catalog has been selected, our database window will display all table.columns that are found in the selected catalog. Our database configuration options will allow users to define:

1. Whether to read each column as a series or a vector
2. Whether or not to include SQL Views along with Tables in the object listing
3. What hard coded frequency value to use when making a series
4. What hard coded start value to use when making a series
5. How many rows to return (all, or a fixed number)

The complete source code for this example is provided in the EdxSamples project available at http://www.eviews.com/EViews8/Enterprise/EDXeg.html.

Create the SQL folder

Right-click the "EdxSamples" project, then select "Add", then "New Folder". Name the folder "SQL".

Create the Database Manager class

Right-click the "SQL" folder and select "Add", then "Class…" from the menu. Name the class "GenericSqlDbMgr.vb" and click Add.

Add the following header lines in the new class:

```
Imports System.Runtime.InteropServices
Imports System.Data.SqlClient

<Guid("XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX"), _
 ClassInterface(ClassInterfaceType.None), _
 ComVisible(True)> _
Public Class GenericSqlDbMgr
```

We also need to make this class implement the IDatabaseManager interface:

```
Public Class GenericSqlDbMgr
    Implements EViewsEdx.IDatabaseManager
```

We also need the following member constant:

```
Public Const TABLE_COLUMN_DELIM As String = "."
```

GetAttributes

```
Public Function GetAttributes(ByVal clientInfo As String) As Object _

        Implements EViewsEdx.IDatabaseManager.GetAttributes
    Return "name=GenericSqlEDX, " & _
           "description=Generic Sql Server EDX Database, " & _
           "type=gsqledx, server, dbids, dbidlabel=Catalog, " & _
           "login=server|user|pass|dbid, search=all|attr, " & _
           "nocreate, readonly"
End Function
```

Since this is a server-based database extension, we include the attribute "`server`". We also include the attribute "`login=server|user|pass|dbid`" to specify which fields the user will need to specify when opening a connection to our database. Our database Ids will consist of SQL Server catalogs so we use the "`dbidlabel=Catalog`" attribute to tell EViews how to label database Id fields within the EViews user interface. We also include the "`dbids`" attribute to tell EViews that our code can provide a list of all available database Ids (via `GetDatabaseIds`).

GetDatabaseIds

When the user clicks the "Browse" button on the Open Database dialog, we will need to provide a listing of the valid catalogs in the currently selected SQL Server installation. We will need the server, username, and password that have already been typed in by the user in the dialog so we can login to the server and get the catalog list:

```
Public Function GetDatabaseIds(ByVal server As String, _
                               ByVal username As String, _
                               ByVal password As String) As Object _
        Implements EViewsEdx.IDatabaseManager.GetDatabaseIds

    'build the connection string
    Dim lsCS As String = "Data Source=" & server & ";"
    If username > "" Then
        lsCS &= lsCS & "User Id=" & username & ";"
    End If
    If password > "" Then
        lsCS &= lsCS & "Password=" & password & ";"
    End If

    'get the list of catalogs
    Dim lsReturn As String = ""
    Dim conn As SqlConnection = Nothing
    Try
        conn = New SqlConnection(lsCS)
        conn.Open()

        Dim d As New SqlCommand("exec sp_databases", conn)
        Dim r As SqlDataReader = d.ExecuteReader()
        Dim lsName As String
        While r.Read
            lsName = Util.myCStr(r(0))
            If lsName > "" Then
```

```
                If lsReturn > "" Then
                    lsReturn &= vbCrLf
                End If
                'on each line, EViews is expecting:
                'code[tab]parentCode[tab]shortDesc[tab]longDesc
                lsReturn &= lsName
            End If
        End While
        r.Close()
        conn.Close()
    Catch ex As Exception
        If InStr(ex.Message, "Login failed for user", _
                 CompareMethod.Text) > 0 Then
            Throw New COMException("",
EViewsEdx.ErrorCode.SECURITY_LOGIN_INVALID)
        Else
            Throw New COMException(ex.Message)
        End If
    Finally
        If conn IsNot Nothing Then
            conn.Close()
        End If
    End Try

    Return lsReturn
End Function
```

For our example we return a multiline string to EViews containing one databaseId on each line. See the API documentation of GetDatabaseIds for a discussion of how to return a more descriptive list of databaseIds, including how to arrange the ids into a hierarchical tree.

myCStr

Add the `myCStr` function to the Util class:

```
Public Shared Function myCStr(ByRef roValue As Object) As String
   Try
      If roValue Is DBNull.Value Then
         Return ""
      End If
      Return CStr(roValue)
   Catch ex As Exception
      Return ""
   End Try
End Function
```

Register the Database Manager class

Type the following into the EViews command window:

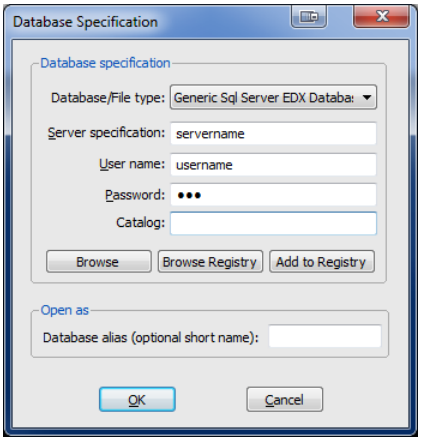
```
edxadd EdxSamples.GenericSqlDbMgr
```

Test the Database Manager class

Now, when the user types:

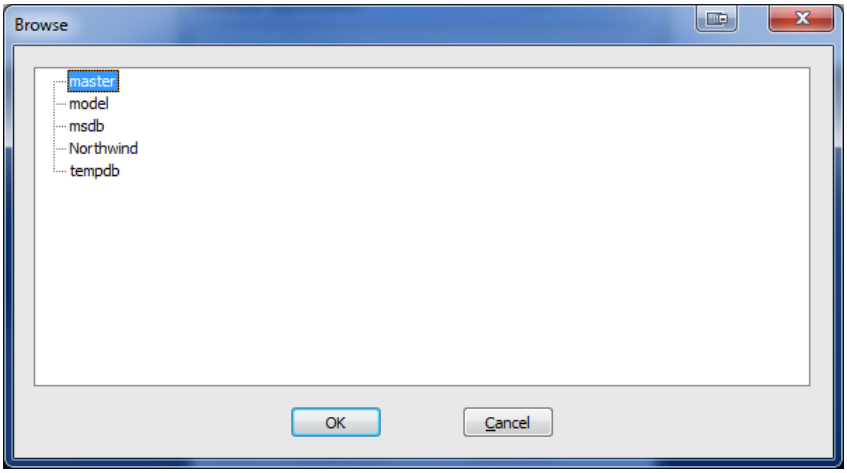
```
dbopen(type=gsqledx)
```

EViews provides this dialog to be filled out:



Once they've typed in the server, username, password, and selected a catalog name, clicking the Browse button will bring up the Browse dialog.

Note that if the login fails, we need to throw the `SECURITY_LOGIN_INVALID` error code so that EViews can display another login dialog to allow the user to correct the login information.



When OK is clicked in the Browse dialog, EViews will call `OpenDb` to create the database connection.

OpenDb

```
Public Function OpenDb(ByVal databaseId As String, _
                       ByVal oc_mode As EViewsEdx.OpenCreateMode, _
                       ByVal rw_mode As EViewsEdx.ReadWriteMode, _
                       ByVal server As String, _
                       ByVal username As String, _
                       ByVal password As String) As EViewsEdx.IDatabase _
```

```
        Implements EViewsEdx.IDatabaseManager.OpenDb
    'build the connection string
    Dim lsCS As String = "Data Source=" & server & ";"
    Dim lsCatalog As String = ""
    Dim lsTable As String = ""
    Dim liPos As Integer = InStr(databaseId, TABLE_COLUMN_DELIM)

    lsCatalog = databaseId
    If lsCatalog > "" Then
        lsCS &= lsCS & "Initial Catalog=" & lsCatalog & ";"
    End If
    If username > "" Then
        lsCS &= lsCS & "User Id=" & username & ";"
    End If
    If password > "" Then
        lsCS &= lsCS & "Password=" & password & ";"
    End If

    'make sure we can connect...
    Dim conn As SqlConnection = Nothing
    Try
        conn = New SqlConnection(lsCS)
        conn.Open()

    Catch ex As Exception
        If InStr(ex.Message, "Login failed for user", _
                CompareMethod.Text) > 0 Then
            Throw New COMException("",
EViewsEdx.ErrorCode.SECURITY_LOGIN_INVALID)
        Else
            Throw New COMException(ex.Message)
        End If
    End Try

    'connection made...
    Return New GenericSqlDb(lsCS, conn, lsTable, mGenSqlPrefs)
End Function
```

Once we build the connection string and verify we can open the SqlConnection object, we pass it to a new instance of the GenericSqlDb class.

Supporting User Configurable Options

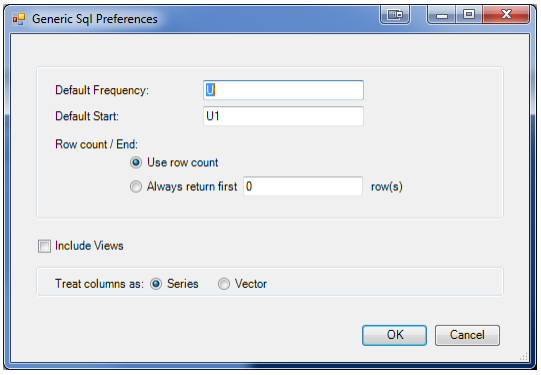
Before we can code the GenericSqlDb class, we'll need to code the user configurable options. This database extension has quite a few options that need to be set by the user in order to operate correctly. First we'll create a class to keep these option values in memory. Create a class and name it GenericSqlPrefs.vb:

```
Public Class GenericSqlPrefs

    Private mbDisplay As Boolean
    Private msDefaultFreq As String
    Private msDefaultStart As String
    Private mbUseRowCount As Boolean
    Private miFixedRowCount As Integer
```

```
Private msIDColumnName As String
Private mbExactMatch As Boolean
Private mbIncludeViews As Boolean
Private mbTreatAsSeries As Boolean

Public Sub New()
    mbDisplay = False
    msDefaultFreq = "U"
    msDefaultStart = "U1"
    mbUseRowCount = True
    miFixedRowCount = 0
    msIDColumnName = ""
    mbExactMatch = False
    IncludeViews = False
    mbTreatAsSeries = True
End Sub

Public Property DisplayOnEachRead() As Boolean
    Get
        Return mbDisplay
    End Get
    Set(ByVal value As Boolean)
        mbDisplay = value
    End Set
End Property

Public Property DefaultFrequency() As String
    Get
        Return msDefaultFreq
    End Get
    Set(ByVal value As String)
        msDefaultFreq = value
    End Set
End Property

Public Property DefaultStart() As String
    Get
        Return msDefaultStart
    End Get
    Set(ByVal value As String)
        msDefaultStart = value
    End Set
End Property

Public Property UseRowCount() As Boolean
    Get
        Return mbUseRowCount
    End Get
    Set(ByVal value As Boolean)
        mbUseRowCount = value
    End Set
End Property

Public Property FixedRowCount() As Integer
    Get
        Return miFixedRowCount
    End Get
```

```
        Set(ByVal value As Integer)
            miFixedRowCount = value
        End Set
    End Property

    Public Property IDColumnName() As String
        Get
            Return msIDColumnName
        End Get
        Set(ByVal value As String)
            msIDColumnName = value
        End Set
    End Property

    Public Property ExactMatch() As Boolean
        Get
            Return mbExactMatch
        End Get
        Set(ByVal value As Boolean)
            mbExactMatch = value
        End Set
    End Property

    Public Property IncludeViews() As Boolean
        Get
            Return mbIncludeViews
        End Get
        Set(ByVal value As Boolean)
            mbIncludeViews = value
        End Set
    End Property

    Public Property TreatAsSeries() As Boolean
        Get
            Return mbTreatAsSeries
        End Get
        Set(ByVal value As Boolean)
            mbTreatAsSeries = value
        End Set
    End Property
End Class
```

Now, create a new Dialog Form in the "SQL" folder and name it GenericSqlPreferences. Add a few controls to make the dialog look similar to:



If you wish to match the control names with those we used in the code below, our naming convention is as follows: The first groupbox contains three edit fields named txtFreq, txtStart, and txtFixedRowCount, and two radio buttons with the names rbUseRowCount and rbAlwaysReturn. The second groupbox contains the rbSeries and rbVector radio buttons. We have also added a checkbox named cbxIncludeViews.

The code behind for this form will look like this:

```
Public Class GenericSqlPreferences

    Private mGenSqlPrefs As GenericSqlPrefs

    Public Sub New(ByRef rPrefs As GenericSqlPrefs)
        ' This call is required by the Windows Form Designer.
        InitializeComponent()

        ' Add any initialization after the InitializeComponent() call.
        mGenSqlPrefs = rPrefs

        With mGenSqlPrefs
            txtFreq.Text = mGenSqlPrefs.DefaultFrequency
            txtStart.Text = mGenSqlPrefs.DefaultStart
            If mGenSqlPrefs.UseRowCount Then
                rbUseRowCount.Checked = True
            Else
                rbAlwaysReturn.Checked = True
            End If
            txtFixedRowCount.Text = mGenSqlPrefs.FixedRowCount.ToString
            cbxIncludeViews.Checked = mGenSqlPrefs.IncludeViews
            If mGenSqlPrefs.TreatAsSeries Then
                rbSeries.Checked = True
            Else
                rbVector.Checked = True
            End If
        End With
    End Sub

    Private Sub OK_Button_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles OK_Button.Click
        'confirm the entered values...
        If Trim(txtFreq.Text) = "" Then
            MsgBox("Default frequency is required.", MsgBoxStyle.Critical)
            Return
```

```
        End If

        If Trim(txtStart.Text) = "" Then
            MsgBox("Default start is required.", MsgBoxStyle.Critical)
            Return
        End If

        If rbAlwaysReturn.Checked Then
            If Util.myCInt(txtFixedRowCount.Text) <= 0 Then
                MsgBox("You must specify a positive row count to always
return.", MsgBoxStyle.Critical)
                Return
            End If
        End If

        'save all the values...
        With mGenSqlPrefs
            .DefaultFrequency = txtFreq.Text
            .DefaultStart = txtStart.Text
            .UseRowCount = rbUseRowCount.Checked
            .FixedRowCount = Util.myCInt(txtFixedRowCount.Text)
            .IncludeViews = cbxIncludeViews.Checked
            .TreatAsSeries = rbSeries.Checked
        End With

        Me.DialogResult = System.Windows.Forms.DialogResult.OK
        Me.Close()
    End Sub

    Private Sub Cancel_Button_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Cancel_Button.Click
        Me.DialogResult = System.Windows.Forms.DialogResult.Cancel
        Me.Close()
    End Sub
End Class
```

Add a new member variable in GenericSqlDbMgr that will keep these options in memory:

```
Public Class GenericSqlDbMgr
    Implements EViewsEdx.IDatabaseManager

    Private mGenSqlPrefs As New GenericSqlPrefs
    Public Const TABLE_COLUMN_DELIM As String = "."
```

Now put the following code in `ConfigurePreferences`:

```
Public Function ConfigurePreferences(ByVal server As String, _
                                     ByVal username As String, _
                                     ByVal password As String, _
                                     ByRef prefs As String) As Boolean _
        Implements EViewsEdx.IDatabaseManager.ConfigurePreferences
    Dim frm As New GenericSqlPreferences(mGenSqlPrefs)
    Dim res As System.Windows.Forms.DialogResult = frm.ShowDialog()

    If res = Windows.Forms.DialogResult.Cancel Then
        Return False
```

```
    End If

    'save the preferences as a single string...
    With mGenSqlPrefs
        prefs = "DisplayOnEach=" & .DisplayOnEachRead.ToString & "|" & _
                "DefaultFreq=" & .DefaultFrequency & "|" & _
                "DefaultStart=" & .DefaultStart & "|" & _
                "UseRowCount=" & .UseRowCount.ToString & "|" & _
                "FixedRowCount=" & .FixedRowCount.ToString & "|" & _
                "IDColumnName=" & .IDColumnName & "|" & _
                "ExactMatch=" & .ExactMatch.ToString & "|" & _
                "IncludeViews=" & .IncludeViews.ToString & "|" & _
                "TreatAsSeries=" & .TreatAsSeries.ToString
    End With
    Return True
End Function
```

and `SetPreferences`:

```
Public Sub SetPreferences(ByVal prefs As String) _
        Implements EViewsEdx.IDatabaseManager.SetPreferences
    If Trim(Util.myCStr(prefs)) = "" Then
        Return
    End If

    Dim la() As String = Split(prefs, "|")
    Dim liPos As Integer
    Dim lsName As String
    Dim lsValue As String
    For Each item As String In la
        If item > "" Then
            liPos = InStr(item, "=")
            If liPos > 0 Then
                lsName = Mid(item, 1, liPos - 1)
                lsValue = Mid(item, liPos + 1)
                With mGenSqlPrefs
                    Select Case LCase(lsName)
                        Case "displayoneach"
                            .DisplayOnEachRead = Util.myCBool(lsValue, False)

                        Case "defaultfreq"
                            .DefaultFrequency = Trim(lsValue)

                        Case "defaultstart"
                            .DefaultStart = Trim(lsValue)

                        Case "userowcount"
                            .UseRowCount = Util.myCBool(lsValue, True)

                        Case "fixedrowcount"
                            .FixedRowCount = Util.myCInt(lsValue)

                        Case "idcolumnname"
                            .IDColumnName = lsValue

                        Case "exactmatch"
```

```
                            .ExactMatch = Util.myCBool(lsValue, False)

                        Case "includeviews"
                            .IncludeViews = Util.myCBool(lsValue, True)

                        Case "treatasseries"
                            .TreatAsSeries = Util.myCBool(lsValue, True)
                    End Select
                End With
            End If
        End If
    Next
End Sub
```

Create the Database class

Right click the "SQL" folder in the Solution Explorer and select "Add", then "Class…" from the menu. In the Add New Item dialog, select the "Class" template and rename the file to "GenericSqlDb.vb" and click Add.

Add the following to the header:

```
Imports System.Runtime.InteropServices
Imports System.Data.SqlClient

Public Class GenericSqlDb
```

And also make sure this class implements the IDatabaseManager interface:

```
Public Class GenericSqlDb
    Implements EViewsEdx.IDatabase
```

Also, include the following member variables:

```
Public Class GenericSqlDb
    Implements EViewsEdx.IDatabase

    Private msCS As String
    Private msCatalog As String
    Private mconn As SqlConnection
    Private mcmd As SqlCommand
    Private mreader As SqlDataReader
    Private mbSearchAbort As Boolean
    Private mobjSync As New Object
    Private mGenSqlPrefs As GenericSqlPrefs
    Private msSchema As String
```

The constructor for this class looks like this:

```
Public Sub New(ByVal vsCS As String, _
               ByRef rconn As SqlConnection, _
               ByVal vsCatalog As String, _
               ByRef rPrefs As GenericSqlPrefs)
```

```
    MyBase.New()

    msCS = vsCS
    msCatalog = vsCatalog
    mconn = rconn
    mGenSqlPrefs = rPrefs
    msSchema = ""
End Sub
```

We will also need three helper functions:

```
Private Function BuildTableName(ByVal vsTableName As String) As String
    If msSchema > "" Then
        vsTableName = msSchema & "." & vsTableName
    End If
    Return vsTableName
End Function

Private Sub ExtractTableCol(ByVal vsObjectId As String, _
                            ByRef rsTableName As String, _
                            ByRef rsColumnName As String)
    Dim liPos As Integer = InStr(vsObjectId,
GenericSqlDbMgr.TABLE_COLUMN_DELIM)
    If liPos > 0 Then
        rsTableName = Mid(vsObjectId, 1, liPos - 1)
        rsColumnName = Mid(vsObjectId, liPos +
Len(GenericSqlDbMgr.TABLE_COLUMN_DELIM))
    Else
        rsTableName = ""
        rsColumnName = ""
    End If
End Sub

Private Sub CloseReader()
    If mreader IsNot Nothing Then
        Try
            mreader.Close()
        Catch ex As Exception
            'ignore
        End Try
    End If
End Sub
```

SearchByAttributes

```
Public Sub SearchByAttributes(ByVal searchExpression As String, _
                              ByVal attrNames As String) _
        Implements EViewsEdx.IDatabase.SearchByAttributes
    Dim lsSQL As String

    'clean up
    CloseReader()

    If mGenSqlPrefs.IncludeViews Then
        lsSQL = "select a.* from INFORMATION_SCHEMA.COLUMNS a " & _
```

```
                "inner join INFORMATION_SCHEMA.TABLES b " & _
                "on b.TABLE_CATALOG = a.TABLE_CATALOG " & _
                "and b.TABLE_SCHEMA = a.TABLE_SCHEMA " & _
                "and b.TABLE_NAME = a.TABLE_NAME " & _
                "and b.TABLE_TYPE in ('BASE TABLE', 'VIEW') " & _
                "order by a.TABLE_NAME, a.COLUMN_NAME"
    Else
        lsSQL = "select a.* from INFORMATION_SCHEMA.COLUMNS a " & _
                "inner join INFORMATION_SCHEMA.TABLES b " & _
                "on b.TABLE_CATALOG = a.TABLE_CATALOG " & _
                "and b.TABLE_SCHEMA = a.TABLE_SCHEMA " & _
                "and b.TABLE_NAME = a.TABLE_NAME " & _
                "and b.TABLE_TYPE in ('BASE TABLE') " & _
                "order by a.TABLE_NAME, a.COLUMN_NAME"
    End If

    'open a new reader object
    mbSearchAbort = False
    mcmd = New SqlCommand(lsSQL, mconn)
    mreader = mcmd.ExecuteReader()
End Sub
```

Depending on whether to include SQL Views or not, we generate a SQL script and query the SQL Server database for a list of all table and column names. We keep a reference to the SQLReader object so that we can query this list one by one during the `SearchNext` call.

SearchNext

```
Public Function SearchNext(ByRef objectId As String, _
                           ByRef attr As Object) As Boolean _
        Implements EViewsEdx.IDatabase.SearchNext
    If mreader Is Nothing Then
        Return False
    End If

    'if we're at the end of the reader object, return false
    If Not mreader.Read Then
        mreader.Close()
        Return False
    End If

    'see if our search has been aborted...
    SyncLock mobjSync
        If mbSearchAbort Then
            mbSearchAbort = False
            CloseReader()
            Return False
        End If
    End SyncLock

    msSchema = Util.myCStr(mreader("TABLE_SCHEMA"))
    objectId = Util.myCStr(mreader("TABLE_NAME")) & _
               GenericSqlDbMgr.TABLE_COLUMN_DELIM & _
               Util.myCStr(mreader("COLUMN_NAME"))
```

```
    ReadObjectAttributes(objectId, "", attr)

    Return True
End Function
```

`SearchNext` will ask for the next row in the `mReader` recordset and place both the table name and column name as a single string as the `objectId`.

One thing to note here is that we do have support built-in for cancelling the search loop that EViews is doing. `mbSearchAbort` is a variable that can be set to `True` in the `SearchAbort` method:

SearchAbort

```
Public Sub SearchAbort() Implements EViewsEdx.IDatabase.SearchAbort
    SyncLock mobjSync
        mbSearchAbort = True
    End SyncLock
End Sub
```

Because this method can be called asynchronously by EViews, we need to make sure we synchronize the access to the member variable `mbSearchAbort` by using `SyncLock`. This is also used in `SearchNext` to make sure we have it locked when we check for the `mbSearchAbort` value.

Back in `SearchNext`, since the attributes for the object are identical to those being retrieved by `ReadObjectAttributes`, we just call that method to retrieve them.

ReadObjectAttributes

```
Public Sub ReadObjectAttributes(ByVal objectId As String, _
                                ByVal defaultFreq As String, _
                                ByRef attr As Object) _
        Implements EViewsEdx.IDatabase.ReadObjectAttributes
    Dim lsTableName As String = ""
    Dim lsColumnName As String = ""
    ExtractTableCol(objectId, lsTableName, lsColumnName)

    Dim lconn As New SqlConnection(msCS)
    lconn.Open()
    Try
        Dim lsSQL As String = "select DATA_TYPE " & _
                              "from INFORMATION_SCHEMA.COLUMNS " & _
                              "where TABLE_NAME = " &
Util.AsQuotedString(lsTableName) & _
                              " and COLUMN_NAME = " &
Util.AsQuotedString(lsColumnName)
        Dim cmd As New SqlCommand(lsSQL, lconn)
        Dim lsTypeName As String = Util.myCStr(cmd.ExecuteScalar())
        If lsTypeName = "" Then
            Throw New COMException("",
EViewsEdx.ErrorCode.RECORD_NAME_INVALID)
        End If
        Dim lsType As String = IIf(mGenSqlPrefs.TreatAsSeries, "series",
"vector")
```

```
        If InStr(lsTypeName, "varchar", CompareMethod.Text) > 0 Then
            lsType = IIf(mGenSqlPrefs.TreatAsSeries, "alpha", "svector")
        End If

        attr = "name=" & objectId & ", type=" & lsType
        If mGenSqlPrefs.UseRowCount Then
            lsSQL = "select count(*) as RecCnt from " &
BuildTableName(lsTableName)
            cmd.CommandText = lsSQL
            Dim liRowCnt As Integer = Util.myCInt(cmd.ExecuteScalar())
            attr &= ", obs=" & liRowCnt.ToString
        ElseIf mGenSqlPrefs.FixedRowCount > 0 Then
            attr &= ", obs=" & mGenSqlPrefs.FixedRowCount.ToString
        End If
        If mGenSqlPrefs.TreatAsSeries Then
            If mGenSqlPrefs.DefaultFrequency > "" Then
                attr &= ", freq=" & mGenSqlPrefs.DefaultFrequency
            End If
            If mGenSqlPrefs.DefaultStart > "" Then
                attr &= ", start=" & mGenSqlPrefs.DefaultStart
            End If
        End If
    Finally
        lconn.Close()
    End Try
End Sub
```

`ReadObjectAttributes` determines the appropriate attributes for the selected table/column. If we're treating the column as a series type, we also provide the default frequency and start value as specified in the user configurable options.

ReadObject

```
Public Sub ReadObject(ByVal objectId As String, _
                      ByVal defaultFreq As String, _
                      ByRef attr As Object, _
                      ByRef vals As Object, _
                      ByRef ids As Object) _
        Implements EViewsEdx.IDatabase.ReadObject
    Dim lsTableName As String = ""
    Dim lsColumnName As String = ""
    ExtractTableCol(objectId, lsTableName, lsColumnName)

    Dim lconn As New SqlConnection(msCS)
    lconn.Open()
    Try
        Dim lsSQL As String = "select DATA_TYPE " & _
                              "from INFORMATION_SCHEMA.COLUMNS " & _
                              "where TABLE_NAME = " &
Util.AsQuotedString(lsTableName) & _
                              " and COLUMN_NAME = " &
Util.AsQuotedString(lsColumnName)
        Dim cmd As New SqlCommand(lsSQL, lconn)
        Dim lsTypeName As String = Util.myCStr(cmd.ExecuteScalar())
```

```
        If lsTypeName = "" Then
            Throw New COMException("",
EViewsEdx.ErrorCode.RECORD_NAME_INVALID)
        End If
        Dim lsType As String = IIf(mGenSqlPrefs.TreatAsSeries, "series",
"vector")
        If InStr(lsTypeName, "varchar", CompareMethod.Text) > 0 Then
            lsType = IIf(mGenSqlPrefs.TreatAsSeries, "alpha", "svector")
        End If

        lsSQL = "select count(*) as RecCnt from " &
BuildTableName(lsTableName)
        cmd.CommandText = lsSQL
        Dim liRowCnt As Integer = Util.myCInt(cmd.ExecuteScalar())

        attr = "type=" & lsType
        If mGenSqlPrefs.UseRowCount Then
            attr &= ", obs=" & liRowCnt.ToString
        ElseIf mGenSqlPrefs.FixedRowCount > 0 Then
            attr &= ", obs=" & mGenSqlPrefs.FixedRowCount.ToString
        End If
        If mGenSqlPrefs.TreatAsSeries Then
            If mGenSqlPrefs.DefaultFrequency > "" Then
                attr &= ", freq=" & mGenSqlPrefs.DefaultFrequency
            End If
            If mGenSqlPrefs.DefaultStart > "" Then
                attr &= ", start=" & mGenSqlPrefs.DefaultStart
            End If
        End If

        Dim liArraySize As Integer = liRowCnt
        If Not mGenSqlPrefs.UseRowCount And mGenSqlPrefs.FixedRowCount > 0
Then
            liArraySize = mGenSqlPrefs.FixedRowCount
        End If

        ReDim vals(0 To liArraySize - 1)
        lsSQL = "select top " & liArraySize.ToString & " " & _
                lsColumnName & " from " & BuildTableName(lsTableName)
        cmd.CommandText = lsSQL
        Dim liIndex As Integer = -1
        Dim lreader As SqlDataReader = Nothing
        Try
            If liRowCnt > 0 Then
                lreader = cmd.ExecuteReader()
                While lreader.Read
                    liIndex += 1
                    vals(liIndex) = Util.myCStr(lreader(0))
                End While
            End If
            While liIndex < (liArraySize - 1)
                liIndex += 1
                vals(liIndex) = Nothing
            End While
        Finally
            If lreader IsNot Nothing Then
                lreader.Close()
```

```
            End If
        End Try
    Finally
        lconn.Close()
    End Try
End Sub
```

`ReadObject` looks a lot like `ReadObjectAttributes` but includes the addition of reading the column data for the specified table and inserting them into the `vals` parameter.

ProposeName

Because we display our object names as "table.column", and because the "." is not allowed in EViews object names within EViews, we can implement the `ProposeName` method to suggest a better name:

```
Public Function ProposeName(ByRef objectId As String, _
                            ByVal destFormat As EViewsEdx.DbFormat, _
                            ByVal destInfo As String) As Boolean _
        Implements EViewsEdx.IDatabaseManager.ProposeName

    Select Case destFormat
        Case EViewsEdx.DbFormat.EViewsDatabase
        Case EViewsEdx.DbFormat.EViewsWorkfile
            Dim liPos As Integer = InStr(objectId, _
GenericSqlDbMgr.TABLE_COLUMN_DELIM)
            If liPos > 0 Then
                objectId = Mid(objectId, liPos + 1)
            End If
    End Select
    'always display to the user the new name
    'in case it conflicts with somthing already
    'in the EViews database or workfile
    Return True
End Function
```

When the destination is an EViews database or workfile, this function will just strip off the table name and the period in `objectId`. We return `True` to tell EViews to display this new name to the user in interactive mode. This gives the user a chance to approve or modify the proposed name.

Close

Because our mReader object can be left open, we need to make sure that when our Database object is closed, we also close mReader to release any SQL Server resources that are still being held. We can do this in the Close method.

```
Public Sub Close() Implements EViewsEdx.IDatabase.Close
    Try
        CloseReader()
        mconn.Close()
    Catch ex As Exception
        'ignore
    End Try
End Sub
```

Add the functions myCBool and AsQuotedString to the Util class:

```
Public Shared Function myCBool(ByRef roValue As Object, _
                          Optional ByVal vbDefaultValue As Boolean = False) _
                          As Boolean
   Try
      If roValue Is DBNull.Value Then
         Return vbDefaultValue
      End If
      Return CBool(roValue)
   Catch ex As Exception
      Return vbDefaultValue
   End Try
End Function

Public Shared Function AsQuotedString(ByRef roObject As Object) As String
   Dim lsValue As String = myCStr(roObject)
   lsValue = "'" & Replace(lsValue, "'", "''") & "'"
   Return lsValue
End Function
```

Summary

At this point, you should be able to use this database extension to open a connection to any SQL Server database and read in the data from any table/column.

One way to improve this database extension is to add support for user-specified ID columns during the `ReadObject` method. This would give EViews a chance to automatically determine the frequency of the observations from the set of date identifiers rather than requiring it to be set by the user in the database manager preferences.

Distributing a Database Extension

Distributing a Database Extension to other users of EViews requires several steps:

1) The software that implements your extension must be installed on the user's system
2) Your class that implements the IDatabaseManager interface must be registered with Windows so that EViews will be able to create it using its ProgId.
3) EViews must be made aware of the ProgId of your database manager class.

Installing your Database Extension

Your database extension can be installed anywhere on the user's file system, including under the normal Program Files subdirectory (as long as it's properly registered). If your extension requires a version of .NET Framework to be pre-installed, make sure your installer either checks for this or knows how to install this pre-requisite during the installation. EViews itself does not require .NET Framework so it is not guaranteed to be installed.

Registering your Database Manager

After your extension has been installed onto the target system, you will have to register it with Windows before it can be found and used by programs such as EViews. Many installer utilities (such as Installshield) can help you do this automatically at the end of the installation, but we'll describe the steps for doing this manually in case your utility does not support this.

Registering your component will depend on whether or not it was developed using the .NET Framework.

If your database extension was built using the .NET Framework, you will have to use the .NET REGASM tool to properly register your extension on the target system. Currently, there are two main versions of REGASM, depending on which version of .NET Framework you are dependent on. Frameworks 3.5 and earlier use REGASM under the "C:\Windows\Microsoft.NET\Framework\v2.0.50727" subdirectory. Frameworks 4.0 and later use REGASM under "C:\Windows\Microsoft.NET\Framework\v4.0.30319" subdirectory.

For example, if your extension was written with .NET Framework 4.0, you will have to launch the 4.0 REGASM tool with the following command:

```
C:\Windows\Microsoft.NET\Framework\v4.0.30319\regasm.exe /silent /codebase
"c:\path\to\your\extension\dll"
```

It is important to remember that if the target system is a 64-bit OS, you must register your extension under 64-bit so that 64-bit EViews users can use it as well. .NET provides a 64-bit version of REGASM under the "Framework64" subdirectory. For example:

```
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\regasm.exe /silent /codebase
"c:\path\to\your\extension\dll"
```

Note that registering the same .NET DLL for both 32-bit and 64-bit use will only work if you compiled the project using the "Any CPU" setting within Visual Studio. Otherwise, cross-bit registration will not work

and may require you to separately build and distribute two different DLLs, one for each architecture. Be sure to test your installation on both 32-bit and 64-bit systems to make sure your classes are being registered properly.

All other COM objects (NOT developed with the .NET Framework) should be registered using the standard REGSVR32 tool:

```
C:\Windows\System32\regsvr32.exe /s "c:\path\to\your\dll"
```

On a 64-bit machine, the above command registers the 64-bit version of your DLL. If you also have a separate 32-bit version of your extension, you'll want to register that separately by using the 32-bit version of regsvr32 under the "SysWOW64" subdirectory:

```
C:\Windows\SysWOW64\regsvr32.exe /s "c:\path\to\your\32 bit\dll"
```

Making EViews aware of your Database Manager

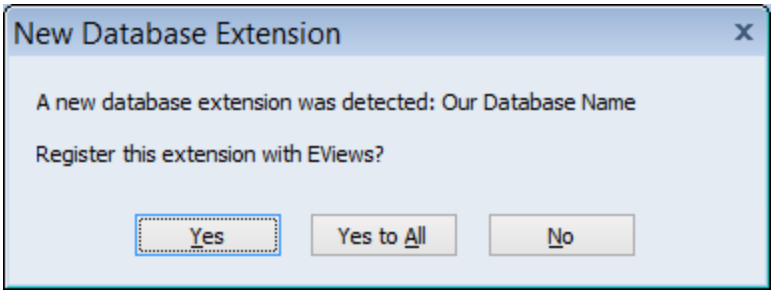
The final step is to make EViews itself aware that your new database extension is available on the target system.

The simplest way to do this is to include an additional empty text file in the installation of your database extension. The file should be written into the subdirectory "%PROGRAMDATA%\IHS EViews\EViews\EDX" and have a file name consisting of the ProgID of your database manager class. The next time EViews runs on the target machine, it will scan the directory, detect the new text file, parse it, then ask the user if they would like to register the new extension within EViews.

The file name may also include a human-readable description enclosed in parenthesis. For example, if a file was written with the name:

```
C:\ProgramData\IHS EViews\EViews\EDX\Company.DatabaseManager (Our Database
Name).txt
```

Then the next time the user launches EViews they will see the following dialog:



Clicking "Yes" will register the extension with EViews. The extension should immediately appear in the list of database types available in the Open Database dialog.

Note that this approach will work even if the user installs EViews AFTER installing your component (in this case, you should create the "%PROGRAMDATA%\IHS EViews\EViews\EDX" folder if it does not already exist during your installation). It will also work if a new user logs onto the target system.

If you would like to pre-register your extension with EViews so that users do not have to confirm the registration upon startup, you can do this by launching EViews with the "/edxaddsilent" command followed by the ProgID of your database manager:

```
(32-bit) C:\Program Files (x86)\EViews 8\EViews8.com /edxaddsilent
Company.DatabaseManager

(64-bit) C:\Program Files\EViews 8\EViews8_x64.com /edxaddsilent Company.DatabaseManager
```

This approach requires that EViews is already installed on the system. Note that this will pre-register your extension for the current user only. If there are other users on the system, you will have to run the edxaddsilent command under each user's account.

EViews also supports several commands inside EViews that allow a user to manage their list of available database extensions. The following commands are available:

```
edxadd progid
```

Enables the database extension with the specified progid.

```
edxdrop progid
```

Disables the database extension with the specified progid.

```
edxscan
```

Forces EViews to rescan the directory of text files representing available extensions, asking the user whether they would like to install any extensions that are not already enabled. This command allows a user to re-enable an extension that has previously been rejected or dropped.

Note that database extension configuration is specific to each user and is stored in the user's EViews32.ini file. You may move the current configuration information to a new machine by simply copying the ini file between the two machines.

API Reference

IDatabaseManager

The Database Manager is the initial contact point with EViews. It provides EViews with meta-data about the database format that it manages (e.g. whether databases in this format are always read-only vs. read/write). It is responsible for creating database objects whenever EViews needs to use a particular database. It also includes functionality for managing entire databases at once such as functions to rename, copy, or delete an entire database.

Methods

AdjustSearchNamePattern	Gives database manager a chance to adjust the name pattern as part of a search by object attributes.
Close	Called by EViews during shutdown.
ConfigurePreferences	Provides a GUI for viewing/editing user specific preferences.
CopyDb	Copies a database.
DeleteDb	Deletes a database.
GetAttributes	Retrieves attributes of this database manager.
GetDatabaseIds	Retrieves a list of database identifiers available for this database format.
OpenDb	Returns an IDatabase interface for a specific database.
ProposeName	Gives database manager a chance to propose a new object name if the current name is illegal in the specified destination.
RenameDb	Renames a database.
SetPreferences	Allows the database manager to realign itself with user preferences set during `ConfigurePreferences`.

IDatabaseManager.AdjustSearchNamePattern Method

Called by EViews to allow the database manager to adjust a name pattern as part of a search by object attributes.

Syntax

Visual Basic (usage)

```
Public Sub AdjustSearchNamePattern(ByRef objectIdPattern As String) _
    Implements EViewsEdx.IDatabaseManager.AdjustSearchNamePattern
```

This function will be called whenever a user performs a search which involves a name pattern. The function gives the database manager a chance to modify the name pattern that was specified by the user before the search begins. For example, the database manager can use this function to make a search for "GDP" be equivalent to a search for "GDP.*". This might make sense in a case where all object identifiers in the database always contain dots so that a simple search for "GDP" will never find any objects.

Note that most database extensions will not require this functionality in which case this function can be left empty.

IDatabaseManager.Close Method

Called by EViews when it no longer needs this database manager (usually during EViews shutdown). Gives the Database Manager a chance to release any allocated resources.

Syntax

Visual Basic (usage)

```
Public Sub Close() Implements EViewsEdx.IDatabaseManager.Close
```

Note: If the Database Manager is being developed in a .NET environment and problems are occurring during EViews shutdown, you might try adding a call to `GC.Collect` during `Close` to encourage .NET to release any COM object references that are being held by dead objects on the managed heap.

IDatabaseManager.ConfigurePreferences Method

Called by EViews when the user selects the View->Preferences menu option on the database window. This function allows the database manager to provide a user interface (a dialog, for example) to configure user specific preferences for the current database.

Syntax

Visual Basic (usage)

```
Public Function ConfigurePreferences(ByVal server As String, _
                                     ByVal username As String, _
                                     ByVal password As String, _
                                     ByRef prefs As String) As Boolean _
    Implements EViewsEdx.IDatabaseManager.ConfigurePreferences
```

The database manager may save these preferences in its own storage (for example, by sending them to the server in a client-server system), or it can return them as a string to EViews by filling out the `prefs` argument. If `prefs` is set and `ConfigurePreferences` returns `True`, EViews will save this string into the EViews INI file for the current user. From then on, every time this Database Manager is loaded, the `prefs` string will be passed back into the manager by a call to the `SetPreferences` function. The `prefs` string can follow any format, although it should generally be kept reasonably short since it will be stored on a single line inside the user's INI file. (Longer information should be stored manually elsewhere on the system.)

IDatabaseManager.CopyDb Method

Called by EViews when the user copies an entire database (using Proc->Copy the Database or the dbcopy command).

Syntax

Visual Basic (usage)

```
Public Function CopyDb(ByVal srcDatabaseId As String, _
                       ByVal destDatabaseId As String, _
                       ByVal server As String, _
                       ByVal username As String, _
                       ByVal password As String, _
                       ByVal overwrite As Boolean) As Boolean _
    Implements EViewsEdx.IDatabaseManager.CopyDb
```

Return `True` to indicate that your function has copied the database, `False` to indicate that it has not. Note that for file-based databases there is a default implementation (when `False` is returned) that will copy all files associated with the database format as reported by the `EXT` or `EXTLIST` attributes in `GetAttributes`.

See the `OpenDb` function for a discussion of the `databaseId`, `server`, `username` and `password` arguments.

Recommended Exceptions:

FILE_FILENAME_INVALID: if the file/database does not exist

FILE_PATHNAME_INVALID: if the path in a filename does not exist

FILE_FILENAME_IN_USE: if the file/database already exists and overwrite is False

SECURITY_LOGIN_INVALID: if a required username or password was missing or invalid

FOREIGN_SERVER_INVALID: if the server specification is invalid

IDatabaseManager.DeleteDB Method

Called by EViews when the user deletes an entire database (using Proc->Delete the Database or the dbdelete command).

Syntax

Visual Basic (usage)

```
Public Function DeleteDb(ByVal databaseId As String, _
                         ByVal server As String, _
                         ByVal username As String, _
                         ByVal password As String) As Boolean _
    Implements EViewsEdx.IDatabaseManager.DeleteDb
```

Return `True` to indicate that your function has deleted the database, `False` to indicate that it has not. Note that for file-based databases there is a default implementation (when `False` is returned) that will delete all files associated with the database format as reported by the `EXT` or `EXTLIST` attributes in `GetAttributes`.

See the OpenDb function for a discussion of the `databaseId`, `server`, `username` and `password` arguments.

Recommended Exceptions:

FILE_FILENAME_INVALID: if the file/database does not exist

FILE_PATHNAME_INVALID: if the path in a filename does not exist

SECURITY_LOGIN_INVALID: if a required username or password was missing or invalid

IDatabaseManager.GetAttributes Method

Called by EViews once per session when the Database Manager is initially loaded. The `GetAttributes` function returns important characteristics of the database format to EViews so that it knows how to interact with this database.

Syntax

Visual Basic (usage)

```
Public Function GetAttributes(ByVal clientInfo As String) As Object _
    Implements EViewsEdx.IDatabaseManager.GetAttributes
```

See Appendix B for a detailed list of attributes and a discussion of how to return them.

The `clientInfo` argument is a string describing the client program that created the manager. EViews will currently always report its `clientInfo` as "EViews8 2014-09-09 (pid=18196)" (where 8 is the current version number, "2014-09-09 is the build date, and pid= is the process id). This parameter can safely be ignored for now: it is designed for future use.

IDatabaseManager.GetDatabaseIds Method

Called by EViews when the user tries to Browse the databases available from this database manager (for example, in the Open Database dialog). This function is typically used by server-based databases to provide a list of identifiers for databases available on the server (where the list may be filtered based on the identity of the user). File-based databases generally do not implement this function, since they can rely on the standard EViews file browsing interface instead.

Syntax

Visual Basic (usage)

```
Public Function GetDatabaseIds(ByVal server As String, _
                               ByVal username As String, _
                               ByVal password As String) As Object _
    Implements EViewsEdx.IDatabaseManager.GetDatabaseIds
```

This function will only be called if the database manager has returned the "dbids" flags in `GetAttributes`.

Database information can be returned to EViews in several formats: as a two dimensional array of strings, as a two dimensional array of variants, or as a single string containing a table in "tsv" format (tabs between fields, linefeeds between lines). The array should have the form:

Id1	parentId1	shortDescription1	longDescription1
Id2	parentId2	shortDescription2	longDescription2
…	…	…	…

where the array contains one row for each available database.

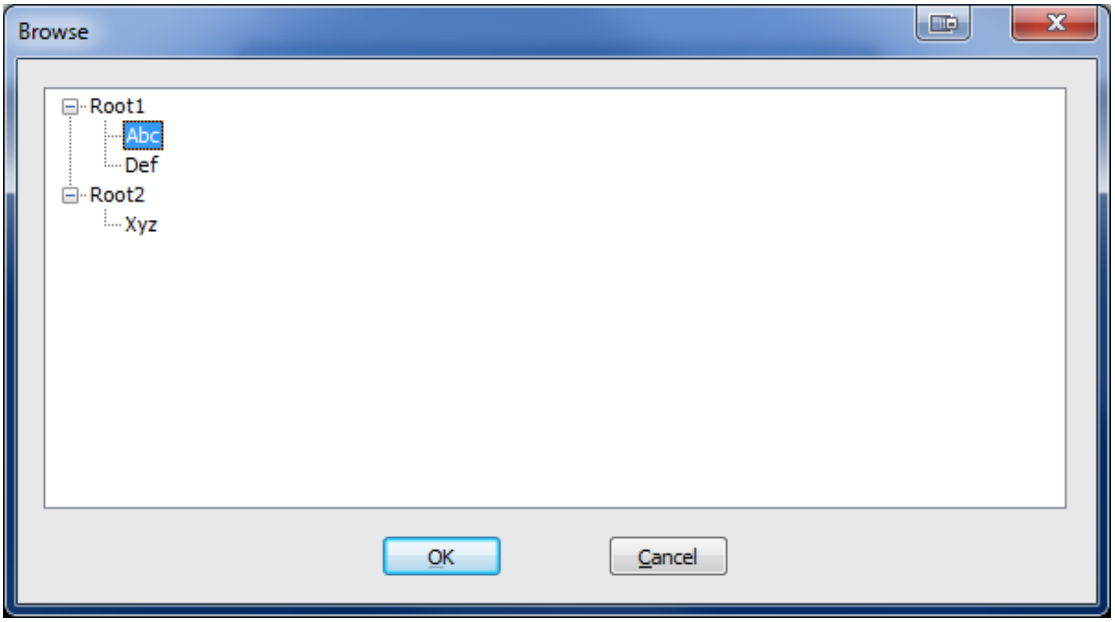
Only the first column is required. A smaller number of columns can be returned when less descriptive information is available.

The first column should contain the string which would need to be used as the `databaseId` argument in `OpenDb` if the user wanted to open the database.

The second column, `parentId`, can be left blank unless you would like the set of database identifiers to be displayed in a hierarchical tree. In this case, you should add extra rows to the array for any non-terminal nodes in the tree and then use the `parentId` column to indicate the parent of each node. For example, the array:

Root1	
Abc	Root1
Def	Root1
Root2	
Xyz	Root2

will display as:



Note that the user will only be able to select terminal nodes as database identifiers.

IDatabaseManager.OpenDb Method

Called by EViews to open a new connection to the specified database. This method must return a reference to an object that implements the IDatabase interface.

Syntax

Visual Basic (usage)

```
Public Function OpenDb(ByVal databaseId As String, _
                       ByVal oc_mode As EViewsEdx.OpenCreateMode, _
                       ByVal rw_mode As EViewsEdx.ReadWriteMode, _
                       ByVal server As String, _
                       ByVal username As String, _
                       ByVal password As String) As EViewsEdx.IDatabase _
    Implements EViewsEdx.IDatabaseManager.OpenDb
```

Databases can be divided into two broad categories: local file databases and client server databases. The arguments to `OpenDb` have slightly different meanings depending on which case you are working with.

For file-based databases, the `databaseId` argument will typically be the full path for the file containing the database. This argument should be sufficient to identify the database to be opened. The server argument is not necessary in this case and can be ignored.

For server-based databases, a variety of information may be necessary to open a database depending on the type of database that your manager supports. You can specify what information is required by your database manager using the `server`, `user`, `pass` and `dbid` flags in the `login` attribute returned by `GetAttributes` (see Appendix B for details). This determines which fields EViews will prompt the user

for when opening a database. For the most part, the meaning of these fields is defined entirely by the specific database manager, but you may like to follow some common guidelines.

The `server` argument will typically contain whatever information is necessary to identify a specific server that supports your database format. It will often be a URL, but it may be some other private form of identifier. If there is only one server for your format, you may prefer to drop the `server` flag from the `login` attribute and hard code all server information within your database extension so that the user need not specify any server information when connecting to your database.

In a server-based system, the `databaseId` can be defined in any way that makes sense for the database system being connected to. If there is only one namespace for all objects available on the server, simply drop the `dbid` flag from the `login` attribute of the manager. This will prevent EViews from prompting the user for a value for `databaseId` when opening the database.

When objects in the server are arranged into more than one namespace, the `databaseId` will typically be used to indicate which namespace should be used by this connection. "Databases", "banks", "catalogs", "subdirectories" and "tables" are all examples of terms that are sometimes used to describe namespaces within a server. You should generally choose a definition of `databaseId` so that the names of objects within a database will be as simple as possible. By specifying extra information within the `databaseId` when the database is opened, a user may be able to use shorter object identifiers when referring to particular objects within the database when working inside EViews.

Because `databaseId` can be used in many different ways, we allow you to change the label that is displayed next to the field within EViews using the `dbidlabel` attribute. You can use this attribute to re-label the field so that it appears more familiar to your users when working with your data. For example, if server namespaces are typically referred to as "Banks" within your system, you may include "`dbidlabel=Bank`" in your attributes to make the EViews interface more intuitive to your users.

The `oc_mode` and `rw_mode` specify how the database is to be opened. The values of `oc_mode` are:

`FileOpen`	Open an existing database, error if the database doesn't already exist
`FileCreate`	Create a new database, error if the database already exists
`FileOpenCreate`	Open an existing database, create a new database if the database doesn't exist
`FileOverwrite`	Delete any existing database, then create a new empty database

The values of `rw_mode` are:

`FileReadOnly`	The database is being opened for reading only. It will not be modified.
`FileReadWrite`	The database is being opened for writing as well as reading.

You may use the `nocreate` and `readonly` flags in the database manager attributes to tell EViews that your format does not support creating new databases or cannot open databases for writing. EViews will never call your database manager with modes that you have indicated that you do not support. All errors will be handled automatically by EViews.

Recommended Exceptions:

FILE_ACCESS_DENIED: if the file/database exists but the user does not have permission to access it

FILE_LOCK_UNAVAILABLE: if the file/database exists, but another user already has it open and is preventing this user from accessing the file

FILE_FILENAME_INVALID: if the file/database does not exist

FILE_PATHNAME_INVALID: if the path in a filename does not exist

FILE_FILENAME_IN_USE: if the file/database already exists and EViews asked to create it

SECURITY_LOGIN_INVALID: if a required username or password was missing or invalid

FOREIGN_SERVER_INVALID: if the server specification is invalid

FOREIGN_CONFIGURATION_REQUIRED: if user preferences must be configured before any databases can be opened (typically when the user first uses the database format)

IDatabaseManager.ProposeName Method

Called by EViews to give the Database Manager a chance to propose a new name for the specified object in cases where the existing name is illegal in the destination format.

Syntax

Visual Basic (usage)

```
Public Function ProposeName(ByRef objectId As String, _
                            ByVal destFormat As EviewsEdx.DbFormat, _
                            ByVal destFreqInfo As String) As Boolean _
    Implements EViewsEdx.IDatabaseManager.ProposeName
```

By default, when an object with an illegal name is copied into an EViews workfile or database, any illegal characters in the name will be replaced with underscores, and the user will be prompted (in interactive mode) to confirm the change.

To change this behavior, simply modify the value of `objectId` to a new value. Return `True` if you would like the user to be given a chance to confirm the proposed name change in interactive mode, `False` if you would like the name change to be applied without a prompt.

The arguments `destFormat` and `destFreqInfo` provide information about the destination container for the object. `destFormat` may contain the following values:

`SameAsSelf`	Destination is this format (the object is being written to this format)
`EViewsDatabase`	Destination is an EViews database (the object is being read from this format)

EViewsWorkfile	Destination is an EViews workfile (the object is being read from this format)

The destFreqInfo argument is currently only used when the destination format is an EViews workfile. It provides information about the frequency, start and end date of the workfile. This may be useful in cases where the object identifier includes frequency information as part of the name and you would like to suppress this when the destination frequency matches the source frequency. See IDatabase::ReadObject for details on the format of the destFreqInfo string.

IDatabaseManager.RenameDb Method

Called by EViews to rename the specified database with a new name.

Syntax

Visual Basic (usage)

```
Public Function RenameDb(ByVal srcDatabaseId As String, _
                         ByVal destDatabaseId As String, _
                         ByVal server As String, _
                         ByVal username As String, _
                         ByVal password As String) As Boolean _
    Implements EViewsEdx.IDatabaseManager.RenameDb
```

Return True to indicate that your function has renamed the database, False to indicate that it has not. Note that for file-based databases there is a default implementation (when False is returned) that will rename all files associated with the database format as reported by the EXT or EXTLIST attributes in GetAttributes.

See the OpenDb function for a discussion of the databaseId, server, username and password arguments.

Recommended Exceptions:

FILE_FILENAME_INVALID: if the file/database does not exist

FILE_PATHNAME_INVALID: if the path in a filename does not exist

FILE_FILENAME_IN_USE: if the file/database already exists and EViews asked to create it

SECURITY_LOGIN_INVALID: if a required username or password was missing or invalid

FOREIGN_SERVER_INVALID: if the server specification is invalid

IDatabaseManager.SetPreferences Method

Gives the Database Manager a chance to restore itself to match preferences previously set by the user during a call to ConfigurePreferences.

Syntax

Visual Basic (usage)

```
Public Sub SetPreferences(ByVal prefs As String) _
    Implements EViewsEdx.IDatabaseManager.SetPreferences
```

These preferences are stored as a single string in the EViews INI file and are passed to the Database Manager immediately after the manager is constructed.

IDatabase

IDatabase is the main interface used by EViews to read and write data objects (e.g. series objects, vectors and strings). It can also be used to copy, rename, and delete objects in the database.

Methods

BeginWrite	Indicate that EViews is beginning a write operation
Close	Release any resources held by the database
CopyObject	Take a copy of an existing object
DeleteObject	Delete an object
DoCommand	Execute a custom command (reserved for future use)
EndWrite	Indicates that EViews is ending a write operation
GetAttributes	Get descriptive information about the database
GetCommandIds	List custom commands (reserved for future use)
ListObjectAttributes	List the set of attributes available for objects in this database
ReadObject	Read an object from the database, including data values
ReadObjectAttributes	Read the attributes of an object in the database, without data values
ReadObjects	Read multiple objects from the database
RenameObject	Rename an object
SearchAbort	Abort a search operation
SearchByAttributes	Initialize a search by object attributes
SearchByBrowser	Initialize a search with a custom browser GUI
SearchNext	Return the next object from an already initialized search
SetAttributes	Set descriptive information about the database
WriteObject	Write an object to the database
WriteObjects	Write multiple objects into the database

IDatabase.BeginWrite Method

Called by EViews to indicate the beginning of a write operation.

Syntax

Visual Basic (usage)

```
Public Sub BeginWrite(ByVal label As String) _
    Implements EViewsEdx.IDatabase.BeginWrite
```

This method is called at the beginning of a write operation that may or may not involve multiple calls to WriteObject. `label` is currently reserved and not used. Can be used to improve the efficiency of subsequent WriteObject calls.

IDatabase.Close Method

Called by EViews when it no longer needs this database to give the database a chance to release resources.

Syntax

Visual Basic (usage)

```
Public Sub Close() Implements EViewsEdx.IDatabase.Close
```

EViews will not make any more calls to the Database class after calling `Close`. In garbage collected environments (.NET), resources such as file handles should be released at this point to avoid resource sharing violations that might be caused by delayed finalization of objects.

IDatabase.CopyObject Method

Called by EViews to make a copy of an existing object within the same database.

Syntax

Visual Basic (usage)

```
Public Sub CopyObject(ByVal srcObjectId As String, _
                      ByRef destObjectId As String, _
                      Optional ByVal overwrite As Boolean = False) _
    Implements EViewsEdx.IDatabase.CopyObject
```

This function will never be called if the database manager has returned the `readonly` attribute.

The `overwrite` flag indicates what should happen if there is an existing object with the name `destObjectId`. If `overwrite` is `True`, the existing object should be deleted and the copy operation should proceed. If `overwrite` is `False`, the function should throw a RECORD_NAME_IN_USE exception to indicate that the copy operation could not proceed.

Note that if this function is not implemented, EViews will attempt to copy the object itself by calling `ReadObject` followed by `WriteObject`.

Recommended Exceptions:

RECORD_NAME_ILLEGAL: if `destObjectId` is not a legal object name

RECORD_NAME_INVALID: if unable to find an object with name `srcObjectId`

RECORD_NAME_IN_USE: if existing object found with name `destObjectId`

IDatabase.DeleteObject Method

Called by EViews to delete an object within the database.

Syntax

Visual Basic (usage)

```
Public Sub DeleteObject(ByVal objectId As String) _
    Implements EViewsEdx.IDatabase.DeleteObject
```

This function will never be called if the database manager has returned the `readonly` attribute.

Recommended Exceptions:

RECORD_NAME_ILLEGAL: if `objectId` is not a legal object name

RECORD_NAME_INVALID: if unable to find an object with name `objectId`

IDatabase.DoCommand Method

Reserved for future use.

Syntax

Visual Basic (usage)

```
Public Function DoCommand(ByVal commandId As String, _
                          ByVal args As Object) As Object _
    Implements EViewsEdx.IDatabase.DoCommand
```

This method is currently not in use.

IDatabase.EndWrite Method

EndWrite method description…

Syntax

Visual Basic (usage)

```
Public Sub EndWrite(ByVal reserved As Integer) _
    Implements EViewsEdx.IDatabase.EndWrite
```

EndWrite method long description…

IDatabase.GetAttributes Method

Called by EViews to obtain information about this particular database.

Syntax

Visual Basic (usage)

```
Public Function GetAttributes() As Object _
    Implements EViewsEdx.IDatabase.GetAttributes
```

There is currently only one database attribute that EViews uses which is the `description` attribute. If a `description` string is returned, it will be displayed by EViews as part of the text of the View -> Database Statistics view from the database window. Attributes are returned in the same way as for the database manager and individual object attributes (see Appendix A for details).

IDatabase.GetCommandIds Method

Reserved for future use.

Syntax

Visual Basic (usage)

```
Public Function GetCommandIds() As Object _
    Implements EViewsEdx.IDatabase.GetCommandIds
```

This method is currently not in use.

IDatabase.ListObjectAttributes Method

Called by EViews to obtain a list of attributes available for the objects in this database.

Syntax

Visual Basic (usage)

```
Public Sub ListObjectAttributes(ByRef attributeList As String, _
                                ByVal delim As String, _
                                ByRef scanForAttributes As Boolean) _
    Implements EViewsEdx.IDatabase.ListObjectAttributes
```

This function can be used to notify EViews which attributes are present within objects in this database. The names of the object attributes should be returned in `attributeList` separated by the delimiter `delim`. The `scanForAttributes` flag is set by EViews on input to indicate whether object specific attributes should be found by scanning the database. If the database has no object specific attributes, or the function scans for these attributes itself, this argument should be set to `False`.

If this function is left empty, EViews will assume that the list of object attributes for this database is the same as the standard list of EViews object attributes in Appendix C.

IDatabase.ReadObject Method

Called to retrieve the attributes, data values, and data identifiers for the specified object.

Syntax

Visual Basic (usage)

```
Public Sub ReadObject(ByVal objectId As String, _
                      ByVal destFreqInfo As String, _
                      ByRef attr As Object, _
                      ByRef vals As Object, _
                      ByRef ids As Object) _
```

```
Implements EViewsEdx.IDatabase.ReadObject
```

EViews calls `ReadObject` whenever a user fetches an object from this database into an EViews workfile or another database.

The `objectId` argument gives the name of the object to read as specified by the EViews user.

The `destFreqInfo` argument provides information about the frequency, start and end date when the destination into which the object is being read is an EViews workfile. This can be used to choose data values at the native frequency in systems where data for an object may exist in the database at more than one frequency. The format of the string is the same as for the `create` command in EViews. See the main EViews documentation for details.

The `attr` argument is used to return metadata about the object. See Appendix B for a discussion of object attributes supported by EViews. If `attr` is not returned, it will be assumed that the object is a series and that date information should be inferred from the `ids` argument (if available).

The `vals` argument is used for returning the data values of the object. The contents of the argument will depend on the type of object being returned. For objects with numeric data (series, scalars, vectors, matrices) the argument may be set to an array of numbers, dates, strings or variants all of which will be converted to double precision values for used inside EViews. Missing values should be coded as NaNs (use "NaN" as the value in .NET code). For objects with character data (alpha series, strings and svectors) the arguments may be set to an array of strings, numbers, dates or variants all of which will be converted to string values for use inside EViews.

The `ids` argument is optional and is provided for the case in which no metadata is available containing frequency information or for the case where there are time gaps between the data values returned in the `vals` array because the database does not include observations for missing data. In these cases, the `ids` argument should be set to an array of date identifiers that match the data values so that EViews can correctly align the data in time. The date identifiers can be provided as an array of COM/.NET date types or as an array of strings that follow a standard EViews date format (eg. "1980Q3", "1980Q4", "1981Q2",…). EViews does not currently support returning text identifiers for matching into undated workfiles that have character id series.

Recommended Exceptions:

RECORD_NAME_ILLEGAL: if `objectId` is not a legal object name

RECORD_NAME_INVALID: if unable to find an object with name `objectId`

FOREIGN_TYPE_READ_UNSUPPORTED: the specified object exists, but cannot be returned to EViews because the object type cannot be converted to an EViews data object

FOREIGN_FREQ_READ_UNSUPPORED: the specified object exists, but it has a frequency that cannot be exported to EViews

FOREIGN_UNSUBSCRIBED_READ: the specified object exists, but this user does not have permission to read the object (typically used for server-based systems where users must pay for access to particular data series)

IDatabase.ReadObjectAttributes Method

Called to retrieve the attributes for the specified object.

Syntax

Visual Basic (usage)

```
Public Sub ReadObjectAttributes(ByVal objectId As String, _
                                ByVal destFreqInfo As String, _
                                ByRef attr As Object) _
    Implements EViewsEdx.IDatabase.ReadObjectAttributes
```

`ReadObjectAttributes` is similar to `ReadObject` but returns less information about the object. It is called by EViews in situations where EViews needs to know some information about the object but does not yet need the full set of data values (for example, in the first stage of exporting one or more series from a database into a new workfile). `ReadObjectAttributes` should only be implemented if the operation can be done more efficiently than reading the entire object. If `ReadObjectAttributes` is not implemented, EViews will call `ReadObject` to retrieve the same information instead.

IDatabase.ReadObjects Method

Called to retrieve attributes, values, and ids for multiple objects specified all at once.

Syntax

Visual Basic (usage)

```
Public Sub ReadObjects(ByVal objectIds As Object, _
                       ByVal destFreqInfo As String, _
                       ByRef attr As Object, _
                       ByRef vals As Object, _
                       ByRef ids As Object) _
    Implements EViewsEdx.IDatabase.ReadObjects
```

This function is not currently used by EViews but is included for future use.

IDatabase.RenameObject Method

Called to rename an object in the database.

Syntax

Visual Basic (usage)

```
Public Sub RenameObject(ByVal srcObjectId As String, _
                        ByVal destObjectId As String) _
    Implements EViewsEdx.IDatabase.RenameObject
```

This function will never be called if the database manager has returned the `readonly` attribute.

Note that if this function is not implemented, EViews will attempt to rename the object itself by calling `CopyObject` followed by `DeleteObject`.

Recommended Exceptions:

RECORD_NAME_ILLEGAL: `destObjectId` is not a legal object name

RECORD_NAME_INVALID: unable to find an object with name `srcObjectId`

RECORD_NAME_IN_USE: existing object found with name `destObjectId`

IDatabase.SearchAbort Method

Called to cancel a search request.

Syntax

Visual Basic (usage)

```
Public Sub SearchAbort() Implements EViewsEdx.IDatabase.SearchAbort
```

Called by EViews when a user cancels an ongoing search operation by hitting the "Esc" key. Indicates that EViews is finished with the current search operation and will not call `SearchNext` again without first initializing a new search.

IDatabase.SearchByAttributes Method

Called to initialize a search across objects in the database to find ones that meet particular criteria.

Syntax

Visual Basic (usage)

```
Public Sub SearchByAttributes(ByVal searchExpression As String, _
                              ByVal attrNames As String) _
    Implements EViewsEdx.IDatabase.SearchByAttributes
```

The `SearchByAttributes` function is the first function call in a sequence of function calls made by EViews when the user performs an operation that requires searching across objects in the database. The `SearchByAttributes` function initializes a new search. Results for the search are then returned one at a time by repeated calls to `SearchNext`. EViews may call `SearchAbort` before reaching the end of the results if the user has cancelled the search by hitting the escape key.

Searches will occur when the user explicitly requests a search from the database window (using the "All", "Easy Query", or "Query" buttons on the database window). They will also occur implicitly in some other cases such as when the user carries out a command that involves wildcard name patterns (eg. "FETCH GDP*").

The `searchExpression` argument is used to tell the database which objects the user is looking for. The contents of the argument will depend on the value of the `SearchAttr` attribute returned by the database manager (see Appendix B). The default value of `SearchAttr` is "Name" in which case `searchExpression` will contain the name pattern that should be matched against `objectIds` when deciding which objects should be returned by `SearchNext`. (Note that name patterns may contain wildcards "*" to match zero or more characters or "?" to match a single character). If `searchAttr` is changed to include other attribute names, `searchExpression` will contain a list of matching values for each field separated by the "|" character. You should only change the default `searchAttr` value if you can efficiently filter objects in the database by fields other than name. This typically requires that indices are available within the database to support fast searching on certain fields.

Note that EViews will always perform an additional filtering of all objects returned by a search to ensure that they match the specified search criterion. This means that a database may return a larger set of objects than what is strictly required and let EViews do the work of discarding any objects that do not match the search criterion. In the most general case, the database can simply return every object in the database on every search and leave everything else to EViews. This may be the best approach for small to medium sized file-based databases where there may be little gain in efficiency in filtering the objects outside of EViews.

The `attrNames` argument provides a list of the attributes that have been requested by the user to be returned as part of the search. This argument is provided for efficiency reasons only. EViews will always extract the particular fields requested by the user from the set of object attributes returned during `SearchNext`, so you are always allowed to return a larger set of attributes than what was requested. This parameter is included for cases where some object attributes may be retrieved much more cheaply than others, in which case you may choose to use the `attrNames` argument to speed up the search process.

IDatabase.SearchByBrowser Method

Called to display a custom database browser window that allows a user to navigate through the objects within the database and select one or more objects.

Syntax

Visual Basic (usage)

```
Public Function SearchByBrowser(ByVal browserArgs As Object, _
                                ByRef attrNames As String) As Object _
    Implements EViewsEdx.IDatabase.SearchByBrowser
```

EViews will only call this function if the database manager has declared that it supports a GUI browser by setting the `search` attribute to include the value `browser` inside `GetAttributes`. In this case, the EViews database window will include a toolbar button marked "Browse" which will call this function whenever it is clicked.

Implementing a custom browser allows a richer user experience in cases where the database is structured in a way that is not exploited by the standard EViews searching capabilities. For example, the objects in the database may be arranged by categories such as country or industry, and users may prefer to navigate through the database using a tree structure based on these attributes when they are searching for a particular data series.

EViews supports two different implementations of a custom browser window: a blocking implementation and a non-blocking implementation.

The blocking implementation is much simpler to implement. In a blocking implementation the database extension simply displays a dialog or form containing the browser interface inside the `SearchByBrowser` function and does not return from the function until the user has closed the dialog or form. In this case, this function should return an empty object ("Nothing" in .NET). EViews will immediately retrieve the items selected by the user by repeatedly called the SearchNext function.

The non-blocking implementation is considerably more complicated. In a non-blocking implementation, the database extension creates an object that implements the browser interface inside the `SearchByBrowser` function, but then immediately returns from the function before the browsing operation is complete. Because control is returned to EViews, this implementation requires that a conversation be established between EViews and the browser so that actions can be coordinated between them. In this case, the `SearchByBrowser` function should return an object that implements the interface EViewsEdx.IDatabaseBrowser and also supports the event source interface EViewsEdx.IDatabaseBrowserEvents. These two interfaces provide two-way communication between EViews and the external browser control. (See the documentation of these interfaces for details).

A non-blocking implementation may either display the control itself, or return an ActiveX control to EViews. In the latter case, EViews will display the control within an MDI window. Note that a User

Control developed in .NET will generally support the necessary ActiveX interfaces for EViews to host the control.

A custom browser window returns results to EViews through the same mechanism as `SearchByAttributes`. EViews will make repeated calls to `SearchNext` retrieving results for a single object with each call.

In the case of a non-blocking implementation (indicated by the function returning an object), EViews will retrieve the results from the browser when the browser signals to EViews that it is ready by sending a browser event such as AddSelected over the IDatabaseBrowserEvents event interface. See documentation of the interface for a list of other events that the custom browser can send to EViews.

The `browserArgs` argument allows EViews to communicate information to the external browser on launch. It is not currently used.

The `attrNames` argument is used to return the list of attributes that will be available for objects selected by the user inside the browser control when browsing is complete. It should be set to contain a comma separated list of object attribute names.

IDatabase.SearchNext Method

Called to return the next object found during a previously initialized search.

Syntax

Visual Basic (usage)

```
Public Function SearchNext(ByRef objectId As String, _
                           ByRef attr As Object) As Boolean _
    Implements EViewsEdx.IDatabase.SearchNext
```

This function is called repeatedly by EViews to retrieve the results of a search operation initialized by a call to `SearchByAttributes` or `SearchByBrowser`.

The function should return the name of the object matching the search criteria in `objectId`, and all other attributes of the object in `attr` (see Appendix C for a discussion of object attributes).

See SearchByAttributes for a discussion of which objects should be returned by this function during a search.

The function should return `True` if an object was found, `False` if there are no more objects to return.

IDatabase.SetAttributes Method

Called to adjust the attributes of a database.

Syntax

Visual Basic (usage)

```
Public Sub SetAttributes(ByVal attr As Object) _
    Implements EViewsEdx.IDatabase.SetAttributes
```

This function is not currently used.

IDatabase.WriteObject Method

Called to store an object into the database.

Syntax

Visual Basic (usage)

```
Public Sub WriteObject(ByRef objectId As String, _
                       ByVal attr As Object, _
                       ByVal vals As Object, _
                       ByVal ids As Object, _
                       ByVal overwriteMode As EViewsEdx.WriteType) _
    Implements EViewsEdx.IDatabase.WriteObject
```

This function will never be called if the database manager has returned the `readonly` attribute.

EViews will call `WriteObject` whenever a user copies an object from either an EViews workfile or another database into this database.

`WriteObject` is the main function used to save data into an external database. The object to be saved is sent to the function in several parts.

The `objectId` contains the name of the object. You should always check whether this name is legal within your external database and throw a RECORD_NAME_ILLEGAL exception in cases where it is not. This exception signals to EViews that it should give the user a chance to correct the name rather than simply failing on the entire write operation.

The `attr` argument contains the set of attributes of the object being saved. See Appendix C for a discussion of the format and contents of the object attributes.

The `vals` argument contains an array of data values associated with the object. Depending on the object type these may take several forms. For objects with numeric data (series, scalars, vectors, matrices) the `vals` argument will contain an array of double precision numbers. Missing values will be coded as NaNs (use the Double.IsNan() function inside .NET to detect these). For objects with character data (alpha series, strings and svectors) the `vals` argument will contain an array of strings.

The `ids` argument is available only when series objects are written. It provides an explicit date identifier for each element of the `vals` array. The date identifier contains the beginning-of-period date for each

observation of the series. These date identifiers may be ignored in cases where the frequency, start and end date of the series are stored explicitly within the database, but they may be useful in cases where the only calendar information saved to the database is the set of observation dates.

`overwriteMode` is used to indicate what should be done in cases where there is already an existing object with the identifier `objectId`. Currently, EViews will only call this function with two values for `overwriteMode`:

`WriteOverwrite`	Replace the existing object with the current object
`WriteProtect`	Don't replace the existing object (throw exception RECORD_NAME_IN_USE)

The additional values are provided for future extensions to EViews and can be ignored for now.

Recommended Exceptions:

RECORD_NAME_ILLEGAL: if `objectId` is not a legal object name

RECORD_NAME_IN_USE: if there is already an existing object with name `objectId`

FOREIGN_TYPE_WRITE_UNSUPPORTED: if the external database does not support writing of this type of object

FOREIGN_FREQ_WRITE_UNSUPPORED: if the external database does not support writing an object with this frequency

IDatabase.WriteObjects Method

WriteObjects method description…

Syntax

Visual Basic (usage)

```
Public Sub WriteObjects(ByRef errors As Object, _
                        ByRef objectIds As Object, _
                        ByVal attr As Object, _
                        ByVal vals As Object, _
                        ByVal ids As Object, _
                        ByVal overwriteMode As EViewsEdx.WriteType) _
    Implements EViewsEdx.IDatabase.WriteObjects
```

WriteObjects method long description…

IDatabaseBrowser

This interface is used to communicate user interface events within EViews to a custom browser control implemented by the database extension.

This interface is used only when the database extension implements a custom browser, and the custom browser is implemented as a non-blocking control. See IDatabase::SearchByBrowser for a discussion.

Methods

EViewsEvent	Notify the browser of an EViews GUI Event

IDatabaseBrowser.EViewsEvent Method

Called by EViews to notify a custom database browser of an EViews GUI event.

The following events are currently sent to the browser by EViews

MessageId	Description
Activate	Indicate that the custom browser should prepare for user interaction (only sent to the browser if it is not hosted by EViews)
Exit	Indicate that the custom browser should close itself

Syntax

Visual Basic (usage)

```
Public Function EViewsEvent(ByVal id As EViewsEdx.MessageId, _
                            ByVal commandArgs As Object) As Object _
    Implements EViewsEdx.IDatabaseBrowser.EViewsEvent
```

IDatabaseBrowserEvents

This interface is used to communicate user interface events that occur within a custom browser control to EViews. This interface is used only when the database extension implements a custom browser, and the custom browser is implemented as a non-blocking control. See IDatabase::SearchByBrowser for a discussion.

A custom browser control can use the browser events interface to perform operations that would normally be carried out by the user within the EViews database window. Examples include copy and paste or drag and drop of a series within the database into an EViews workfile window. The custom browser may also use the events interface to hide the main EViews database window so that it becomes the primary user interface for the database.

Using the IDatabaseBrowserEvents class within a .NET project requires the following steps:

1) Set the ComSourceInterfaces attribute of your custom browser class to notify .NET that your browser class will generate database browser events
2) Declare the event you are going to raise
3) Use the RaiseEvent statement to notify EViews that an event has occurred.

The following VB .NET code fragment contains a typical example of how this might look. The example implements a function SendMessage that may be called to notify EViews of an event.

```
<ClassInterface(ClassInterfaceType.None), _

 ComSourceInterfaces(GetType(EViewsEdx.IDatabaseBrowserEvents)), _

 ComVisible(True)> _

Public Class MyCategoryBrowserControl

    Implements EViewsEdx.IDatabaseBrowser

    'declare event used to send messages to EViews

    Public Event BrowserEvent(ByVal commandId As EViewsEdx.MessageId, ByVal
commandArgs As Object, ByRef Result As Object)

    'actual function that sends a message to EViews

    Private Function SendMessage(ByVal commandId As EViewsEdx.MessageId,
Optional ByVal commandArgs As Object = Nothing) As Object

        Dim eventResult As Object = Nothing

        RaiseEvent BrowserEvent(commandId, commandArgs, eventResult)

        Return eventResult

    End Function
```

```
End Class
```

If you are using .NET Framework 4.0 or later, you may experience problems where the .NET Framework does not appear to send raised events to EViews. It is not clear why this problem occurs but it seems to be related to the fact that the event interface is declared within a type library built within a native code project. If you experience this behavior, please add an additional reference to the library EViewsEdxNet.dll to your project and then refer to EViewsEdxNet.IDatabaseBrowserEvents instead of EViewsEdx.IDatabaseBrowserEvents when setting the ComSourceInterfaces attribute. (The EViewsEdxNet type library exports exactly the same interface as the EViewsEdx type library but from within a .NET project instead of a native code project).

Methods

BrowserEvent	Notify EViews of a custom browser GUI Event

IDatabaseBrowserEvents.BrowserEvent Event

Raised by a custom database browser to notify EViews of browser GUI events.

Syntax

Visual Basic (usage)

```
Public Event BrowserEvent(ByVal id As EViewsEdx.MessageId, _
                          ByVal commandArgs As Object, _
                          ByRef result As Object) _
```

The following event notifications may be sent to EViews

MessageId	Description
ClearExisting	Instructs EViews to clear out the current contents of the EViews database window
AddSelected	Asks EViews to add the items selected in the browser to the EViews database window. EViews will retrieve the selected items by repeatedly calling the SearchNext function on the associated database object.
CopySelected	Asks EViews to add the items selected in the browser to the EViews database window and copy them to the clipboard so they may be pasted into an EViews workfile or database. EViews will retrieve the items to copy by repeatedly calling the SearchNext function on the associated database object.
DragSelected	Asks EViews to add the items selected in the browser window to the EViews database window and return data that is required to perform a drag and drop operation of the items within EViews. EViews will retrieve the items to drag by repeatedly calling the SearchNext function on the associated database object. The byte array returned by EViews in the result argument should be wrapped into a MemoryStream and placed into a data object using the custom format "EViewsEdxBrowserSelection". Pass this data object to the DoDragDrop()

	function to begin the drag-drop operation.
DragSelectedAsLink	Same as DragSelected but prepares for a drag-as-link / paste special operation (right mouse button drag operation within EViews).
HideDbWin	Hides the EViews database window. The window will be automatically restored when the custom browser control is closed.
RestoreDbWin	Restores the EViews database window (after hiding with HideDbWin)
PreviewSelected	Asks EViews to open a preview window for the selected items (same as double clicking on an item in the EViews database window). EViews will retrieve the items to preview by repeatedly calling the SearchNext function on the associated database object.
Exit	Requests that EViews close the window hosting the custom database browser

Frequency

Frequency is a utility class implemented by EViews that represents the frequency of a regularly spaced time series. The class exports functionality available inside EViews for working with calendar dates and frequencies which may be useful in implementing an EViews database extension. Use of the Frequency class is optional. It does not appear as an argument of any function in IDatabaseManager or IDatabase.

Note that the Frequency class can represent a simple frequency without a base date such as "Q" for a quarterly frequency, or it may represent a frequency that includes an explicit base date such as "Q 1980" for a quarterly frequency starting in 1980. If a base date is specified, observations within the frequency will be numbered so that the base date falls within observation zero. If no base date is specified, observation numbers will be centered on an arbitrary default base date. Note that the difference between two observation numbers will be the same no matter what base date is used.

Methods

DToO	Convert a calendar date/time into an observation number within this frequency
DToString	Convert a calendar date/time into a string
IsValid	Test whether this frequency has been set to a valid value
OtoD	Convert an observation number within this frequency into a calendar date/time
Set	Initialize this frequency from a text specification
SetFromIds	Initialize this frequency to cover a set of date/time identifiers
SetWithBase	Initialize this frequency from a text specification and a base date
StringToD	Convert a string into a calendar date/time
ToString	Create a text specification representing the current frequency

Frequency.DToO Method

The DtoO method returns the observation number within this frequency that contains the specified date.

Syntax

Visual Basic (usage)

```
Public Function DtoO(dateVal As Date, _
                     Optional isExcluded as Boolean) As Integer
```

If no observation contains the date (because the date is a day of week or a time of day that has been excluded from the frequency) the function will return the first observation number after the date and the isExcluded argument) will be set to true. If a base date has been provided for the frequency, the observation number will be zero for the observation that contains the base date and dates within observations before the base date will return negative observation numbers. If no base date has been provided, observation numbering is arbitrary but differences between observation numbers can still be used to calculate observation counts.

Frequency.DToString Method

The DtoString method returns a string representation of the specified date.

Syntax

Visual Basic (usage)

```
Public Function DtoString(dateVal As Date, _
                          Optional format As String = "") As String
```

If no format is provided, the method returns the date in the format used for an observation label in an EViews workfile of this frequency (e.g. "1980Q3" for the date 1980-07-01 in a quarterly workfile). Alternatively, an explicit format may be specified which should follow the standard EViews date format rules as used by the @datestr() function of EViews. For example, "Wdy Day Month[,] yyyy" would produce "Tue 1st July, 1980" for the date 1980-07-01. When no format is provided and the date string contains ambiguous day/month or month/day fields, EViews will use the bDayBeforeMonth flag passed into the Set() function when the frequency was initialized to determine which field is which.

Frequency.IsValid Method

Tests whether the frequency has been set to a valid value.

Syntax

Visual Basic (usage)

```
Public Function IsValid() As Boolean
```

The frequency will be invalid if no 'Set' function has been called or if the arguments provided in a call to the 'Set' function did not specify a valid frequency.

Frequency.OtoD Method

The OtoD method returns the start or end date/time for the specified observation number within this frequency.

Syntax

Visual Basic (usage)

```
Public Function OtoD(obs As Integer, _
                     Optional bEndDate As Boolean = False) As Date
```

If bEndDate is true, the last millisecond that is included within the observation will be returned. Note that there may be a gap between the end date of one observation and the start date of the next observation in cases where some days of the week or times of day have been excluded from the frequency.

Frequency.Set Method

Initializes the frequency from a text representation of the frequency.

Syntax

Visual Basic (usage)

```
Public Function Set(spec As String, _
                    Optional bDayBeforeMonth As Boolean = False) As Boolean
```

The general form of an EViews frequency specification is:

```
[step]unit[(options)] [basedate]
```

where square brackets indicate optional parts of the specification which may be omitted.

The step field is an integer containing the number of time units per observation. Unit may be one of the following units of time: A=year, S=halfyear, Q=quarter, M=month, T=tenday (third of a month), BM=halfmonth, F=fortnight (two weeks), W=week, D=day, B=business day (Monday to Friday), H=hour, MIN=minute, SEC=second. Most common names for time units are also supported. For example, "A", "Annual", "Y", "Year", "Yearly" are all valid ways of describing a yearly time unit.

For annual, semiannual and quarterly data, the options section may contain a month name to specify a monthly alignment other than January. For example, "A(Jul)" is an annual frequency where each year starts July 1st. For weekly data, options may contain a day of week on which to begin the month. For example, "W(Tue)" is weekly data where the week begins on Tuesday. For daily and intraday frequencies, options may contain a day of week range and a time of day range. For example, "15min(Mon-Fri,9am-5pm)") describes a fifteen minute frequency that excludes Saturdays and Sundays and also excludes overnight times between 5:15pm and 9AM.

If a base date is provided, observations will be numbered so that the specified base date is contained within observation zero. If no base date is provided, a default base date will be used and observations will be numbered from an arbitrary starting point. Note that the base date is used to determine the alignment of the frequency in some cases. For example a frequency of "2A 2000" will have observations beginning in years 2000, 2002 and 2004 while a frequency of "2A 2001" will have observations beginning in years 2001, 2003, 2005. Similarly, "A jul2013" will generate a 'fiscal' annual frequency with years starting in July and "W 1Jan2014" will generate a weekly frequency with weeks starting on Wednesday. Note that you can prevent month and week alignments from following the base date by specifying an explicit alignment option in the frequency such as "A(Jan) jul2013" for an annual frequency beginning on Jan 1st or "W(Mon) 1Jan2014" for a weekly frequency beginning on Monday.

The bDayBeforeMonth flag is used to determine day/month order if a base date is provided using mm/dd/yyyy or dd/mm/yyyy format. The option is not used if a base date is provided in the ISO format YYYY-MM-DD. Note that the value of the flag is remembered by the class and is used for all subsequent string to date or date to string conversions performed by the class.

The function returns true if the frequency was successfully set, or false if the specification was invalid.

Frequency.SetFromIds Method

Initializes the frequency from a set of date/time identifiers associated with the observations of a time series.

Syntax

Visual Basic (usage)

```
Public Function SetFromIds(dateids As Object, _
                    Optional bAllowFiscalAlignment As Boolean = True, _
                    Optional bAllowWeekdayAlignment As Boolean = True, _
                    Optional bDayBeforeMonth As Boolean = False) As Boolean
```

The frequency is chosen so that each different date/time in the set falls within a different observation and the number of missing observations is kept to a minimum. The function will typically do a good job of automatically determining the frequency associated with a set of observations of a time series and is used within EViews to attach frequencies to incoming data in cases where no explicit frequency is provided. The set of dates provided must be either increasing or decreasing.

If bAllowFiscalAlignment is true, EViews may select 'fiscal' frequencies that are aligned on a month other than January for annual, semiannual and quarterly data. If bAllowFiscalAlignment is false, EViews may only select January aligned frequencies.

If bAllowWeekdayAlignment is true, EViews may generate weekly frequencies aligned on any day of the week. If bAllowWeekdayAlignment is false, EViews will only generate weekly frequencies that begin on Monday (and end on Sunday).

The bDayBeforeMonth flag sets whether the class will prefer day/month or month/day order in cases where the ordering of day and month fields is ambiguous.

The function returns true if the frequency was set successfully or false if no frequency could be determined for the set of dates provided.

Frequency.SetWithBase Method

Similar to Set() except that the base date is explicitly provided in a separate argument (rather than as an optional second part within the text specification). See documentation of Set() for a discussion.

Syntax

Visual Basic (usage)

```
Public Function SetWithBase(spec As String, _
```

```
                        BaseDate As Date, _
                        Optional bDayBeforeMonth As Boolean = False) As Boolean
```

Frequency.StringToD Method

Converts a string containing a date/time or year/period into a calendar date value.

Syntax

Visual Basic (usage)

```
Public Function StringtoD(dateStr As String, _
                          Optional format As String = "", _
                          Optional bEndDate As Boolean = False) As Date
```

The function supports explicit period-in-year formats such as "1990Q2" for the second quarter of 1990. The function also supports ambiguous period-in-year formats such as "1990:2" in which case the current frequency is used to determine the meaning of the period number (e.g. February 1990 if the frequency is monthly).

If bEndDate is true, the function returns the last millisecond that is within the specified input. For example, the end date of "2010" will be the last millisecond before midnight on Dec 31st 2010.

If no format is provided, EViews automatically interprets the date following its standard rules. In this case ambiguous day/month formats will be interpreted according to the bDayBeforeMonth flag passed into the Set() function when the frequency class was initialized. (ISO dates of the form YYYY-MM-DD will be unaffected by the flag). Alternatively, an explicit format can be provided which should follow the standard EViews date format specification as used by the EViews function @dateval. For example, the format "[Q]q yyyy" can be used to read a quarterly date such as "Q3 1990".

Frequency.ToString Method

Produces a string representation of the frequency.

Syntax

Visual Basic (usage)

```
Public Function ToString(Optional bIncludeBaseDate As Boolean = True) _
    As String
```

The output string follows the format outlined in the documentation of the Set() function.

If bIncludeBaseDate is true, a base date will be included in the output string if an explicit base date was provided when the frequency was initialized. If bIncludeBaseDate is false, the base date will never be included in the output string.

Note that in some cases the base date is necessary to fully describe the frequency. For example a two-yearly frequency requires a base date to determine whether observations should begin on odd or even numbered years.

JsonReader

JsonReader is a utility class implemented by EViews to assist with reading data in JSON (JavaScript Object Notation) format. JSON is a popular standardized format used for communicating information between different programs such as between Web servers and clients on the web.

The JsonReader class transforms raw JSON content into a sequential stream of tokens. Each token can be a simple scalar value such as a number, string, Boolean or null value, an object fieldname, or a structural marker such as an array or object beginning or end. The class is designed to be suitable for scanning through very large files of JSON data. Use of the JsonReader class is optional. It does not appear as an argument of any function in IDatabaseManager or IDatabase.

A typical use of the JsonReader class follows the following steps:

1) Construct the JsonReader class
2) Call AttachToFile to attach the reader to a disk file or AttachToData to attach the reader to data currently in memory
3) Call the ReadToken function to read through the input one token at a time. Alternatively, you may look at the next token without reading past it by calling the PeekToken function.
4) Call the Detach function to indicate you have finished working with the file or memory

A variety of additional functions are provided to simplify common operations and support moving to different locations within the input stream. Detailed information on these functions is provided below.

Methods

AttachToData	Attach the reader to data currently in memory
AttachToFile	Attach the reader to a disk file
Detach	Detach the reader from the current file or memory
GetPosition	Return the starting position of the next token
PeekToken	Peek at the next JSON token without advancing past it
ReadField	Read a name/value pair representing an object field
ReadToken	Read the next JSON token and advance past it
ReadValue	Read the next value (including an array)
Restart	Restart reading from a new position
RewindToStart	Move backwards to the beginning of the current object or array
RewindValue	Move backwards to the start of the last value that was read
SkipToEnd	Move forwards to the end of the current object or array
SkipToField	Move forwards to the specified field of the current object
SkipValue	Move forwards past the next value (including an object or array)
UnreadToken	Return the last token read back into the input stream

JsonReader.AttachToData Method

The AttachToData method attaches the reader to a piece of data currently in memory.

Syntax

Visual Basic (usage)

```
Public Sub AttachToData(data As Object, Optional options As String = "")
```

A common example of using this function would be to attach to content returned by a web server. The data argument passed into the function may either contain a string or a byte array. If the data is provided as a byte array it may contain UTF-8, UTF-16 little-endian or UTF-16 big-endian data with or without a leading byte order mark.

The options argument may be used to specify a list of comma separated options where each option may either be a simple option name or a name-value pair separated by a colon. The only option currently supported by this function is "string=utf8" which specifies that tokens containing string values should be returned in byte arrays containing UTF-8 encoded data. If this option is not specified, string values will be returned using string objects containing UTF-16 encoded data.

JsonReader.AttachToFile Method

The AttachToFile method attaches the reader to a file.

Syntax

Visual Basic (usage)

```
Public Sub AttachToFile(filename As String, Optional options As String = "")
```

The filename argument should contain the full path of the file. The file must contain data in UTF-8 format and may or may not contain a byte order mark.

The options argument may be used to specify a list of comma separated options where each option may either be a simple option name or a name-value pair separated by a colon. The only option currently supported is "string=utf8" which specifies that tokens containing string values should be returned in byte arrays containing UTF-8 encoded data. If this option is not specified, string values will be returned using string objects containing UTF-16 encoded data.

JsonReader.Detach Method

The Detach method detaches the reader from the current file or data in memory.

Syntax

Visual Basic (usage)

```
Public Sub Detach()
```

Explicitly calling Detach forces any file handles or memory currently allocated by the reader to be released immediately. If Detach is not called, all resources will still be freed eventually but the timing may be unpredictable.

JsonReader.GetPosition Method

The GetPosition method returns a bookmark representing the starting position of the next token.

Syntax

Visual Basic (usage)

```
Public Function GetPosition() As Long
```

The bookmark returned by this function may be passed into the Restart function at a later point to restart reading from the current location.

JsonReader.PeekToken Method

The PeekToken method returns the next token from the input without advancing past it.

Syntax

Visual Basic (usage)

```
Public Function PeekToken(Optional ByRef value As Object = Nothing) As
EViewsEdx.JsonTokenType
```

The PeekToken method is similar to ReadToken except that it leaves the current input position unchanged so that a subsequent call to ReadToken or PeekToken will return the same value.

PeekToken may be useful in cases where you would like your code to branch based on the type of the next token without affecting the current position of the reader within the input data.

JsonReader.ReadField Method

The ReadField method returns the next name/value pair representing a field within an object.

Syntax

Visual Basic (usage)

```
Public Function ReadField(ByRef name As String, ByRef value As Object) As
Boolean
```

If the next token is not a fieldname, the function returns false and does not read past the token.

The ReadField function is intended to simplify reading through the fields of simple non-nested objects. Note that this function cannot read fields where the field values are objects themselves or arrays containing objects since the JsonReader class cannot return a JSON object within a single value.

JsonReader.ReadToken Method

The ReadToken method returns the next token from the input data and advances past it.

Syntax

Visual Basic (usage)

```
Public Function ReadToken(Optional ByRef value As Object = Nothing) As
EViewsEdx.JsonTokenType
```

The following token types and values may be returned by the method.

Token Type	Value	Description
JSON_NO_VALUE	Nothing	Indicates that the reader has not yet been attached to input data
JSON_NULL	Nothing	A JSON null value
JSON_BOOL	Boolean	A Boolean value (true or false)
JSON_NUMBER	Double	A numeric value
JSON_STRING	String/Byte array	A string value
JSON_FIELDNAME	String	An object field name
JSON_ARRAY_BEGIN	Nothing	A marker indicating the beginning of an array
JSON_ARRAY_END	Nothing	A marker indicating the end of an array
JSON_OBJECT_BEGIN	Nothing	A marker indicating the beginning of an object
JSON_OBJECT_END	Nothing	A marker indicating the end of an object
JSON_EOF	Nothing	A marker indicating that the end of the attached content has been reached
JSON_ERROR	String	Indicates that a parsing error has occurred. The value will be set to a descriptive error message.

By default string values will be returned in string objects (containing UTF-16 data). If the 'string=utf8' option was set when attaching to the input data, string values will be returned in byte arrays containing UTF-8 data.

Note that ReadToken is similar to PeekToken except that PeekToken does not advance past the token returned by the function.

JsonReader.ReadValue Method

The ReadValue method returns the next value from the input.

Syntax

Visual Basic (usage)

```
Public Function ReadValue() As Object
```

The function will return an error unless the next token is a scalar value (null, Boolean, number or string) or the beginning of an array.

JsonReader.Restart Method

The Restart method causes the reader to discard any existing information about the input and begin reading from a different location.

Syntax

Visual Basic (usage)

```
Public Sub Restart(Optional position As Long = 0)
```

If the location parameter is zero (the default value) the reader will restart reading from the beginning of the input data. The location parameter may also be a value returned by the GetPosition function, in which case the reader will begin reading from the location it was at when GetPosition was called.

Note that you may only restart to a location where the next token is the beginning of a value (a number, a string, a Boolean value, a null value, the beginning of an array of the beginning of an object). You may not restart reading from a location where the next token is a field name, the end of an array or the end of an object. Furthermore, once you restart at a new location you may only read to the end of the value you have restarted at. Reading beyond the end of this value will generate an error.

JsonReader.RewindToStart Method

The RewindToStart method rewinds to the beginning of the current object or array.

Syntax

Visual Basic (usage)

```
Public Sub RewindToStart()
```

A typical use would be to skip back to the start of an object after reading a single field value from the object following a call to SkipToField.

Note that repeatedly rewinding to the beginning of an object and re-reading it is less efficient than processing the object sequentially.

JsonReader.RewindValue Method

The RewindValue method may be used to rewind reading back to the beginning of the last value read, including to the beginning of an object or array that has just ended.

Syntax

Visual Basic (usage)

```
Public Sub RewindValue()
```

Note that you may only rewind a single value using RewindValue. You may not call the function repeatedly to rewind through multiple values.

Note that repeatedly rewinding to the beginning of an object and re-reading it is less efficient than processing the object sequentially.

JsonReader.SkipToEnd Method

The SkipToEnd method advances to the end of the current object or array.

Syntax

Visual Basic (usage)

```
Public Sub SkipToEnd()
```

A typical use would be to skip to the end of an object after reading a single field from the object by using SkipToField.

JsonReader.SkipToField Method

The SkipToField method advances through the fields of an object until it finds a field that matches the specified field name.

Syntax

Visual Basic (usage)

```
Public Function SkipToField(fieldname As String) As Boolean
```

If a matching field is found, the function returns true and the next token will be the start of the value associated with the field. If no matching field is found, the function returns false and the next token will be the token immediately after the end of the object.

JsonReader.SkipValue Method

The SkipValue method advances past one complete value from the input data without returning it.

Syntax

Visual Basic (usage)

```
Public Sub SkipValue()
```

If the next token is a scalar (a number, string, Boolean or null value) calling the function is equivalent to calling ReadToken once and discarding the token value. If the next token is the beginning of an array or object, SkipValue will read tokens repeatedly until it reaches the end of that array or object.

Calling SkipValue is the recommended method for reading past the value of an unrecognized field within an object.

JsonReader.UnreadToken Method

The UnreadToken method returns the last token read back into the input stream so that the next call to ReadToken or PeekToken will return the same token again.

Syntax

Visual Basic (usage)

```
Public Sub UnreadToken()
```

The UnreadToken function may only be called once immediately after a call to ReadToken.

JsonWriter

JsonWriter is a utility class implemented by EViews to assist with writing data in JSON (JavaScript Object Notation) format. JSON is a popular standardized format used for communicating information between different programs such as between servers and clients on the Web.

The JsonWriter class transforms a sequential stream of output tokens into JSON content. Each token can be a simple scalar value such as a number, string, Boolean or null value, an object fieldname, or a structural marker such as an array or object beginning or end. The JsonWriter will generate an error if the set of output tokens do not represent valid content (for example, if array and object beginnings and ends are not correctly matched).

Use of the JsonWriter class is optional. It does not appear as an argument of any function in IDatabaseManager or IDatabase.

A typical use of the JsonWriter class follows the following steps:

1) Construct the JsonWriter class
2) Call AttachToFile to attach the writer to a disk file or AttachToBuffer to attach the writer to a buffer in memory
3) Call the WriteToken function to add content to the output one token at a time, or call WriteValue to write a single value including an entire array.
4) Call the Flush function to commit any unsaved data into the attached file or to retrieve content from the attached memory buffer
5) Call the Detach function to indicate you have finished working with the file or memory

The JsonWriter class produces output in UTF-8 format.

Methods

AttachToBuffer	Attach the writer to a memory buffer
AttachToFile	Attach the writer to a disk file
Detach	Detach the writer from the current file or memory
Flush	Flush or retrieve any unsaved content
WriteField	Write a name/value pair for a field within an object
WriteToken	Write a token
WriteValue	Write a value

JsonWriter.AttachToBuffer Method

The AttachToBuffer method initializes the writer for writing to a memory buffer.

Syntax

Visual Basic (usage)

```
Public Sub AttachToBuffer(Optional initialBufferSize As Integer = 4096,
Optional options As String = "")
```

The initialBufferSize value is the number of bytes initially allocated to hold the output content. By default, 4096 bytes will be allocated. The buffer will be resized automatically if necessary, but it is more efficient to allocate a large buffer if you are likely to generate a large amount of output.

The options argument may be used to specify a list of comma separated options where each option may either be a simple option name or a name-value pair separated by a colon. Currently supported options are "white=pretty" to indicate that whitespace should be inserted into the output to make the output more human readable and "bom" to indicate that a Unicode byte-order-marker should be inserted at the beginning of the output to indicate the character encoding.

JsonWriter.AttachToFile Method

The AttachToFile method initializes the writer for writing to a file. If there is an existing file it will be overwritten by the call.

Syntax

Visual Basic (usage)

```
Public Sub AttachToFile(filename As String, Optional options As String = "")
```

The options argument may be used to specify a list of comma separated options where each option may either be a simple option name or a name-value pair separated by a colon. Currently supported options are "white=pretty" to indicate that whitespace should be inserted into the output to help make the output human readable and "bom" to indicate that a Unicode byte-order-marker should be inserted at the beginning of the output to indicate the character encoding.

JsonWriter.Detach Method

The Detach method detaches the reader from the current file or data in memory.

Syntax

Visual Basic (usage)

```
Public Sub Detach()
```

Explicitly calling Detach forces any file handles or memory currently allocated by the reader to be released immediately. If Detach is not called, all resources will still be freed eventually but the timing may be unpredictable.

JsonWriter.Flush Method

If the writer is attached to a file, the Flush method writes any unsaved data to disk. If the writer is attached to a memory buffer, the Flush method returns the current content of the buffer as a byte array and clears the buffer.

Syntax

Visual Basic (usage)

```
Public Function Flush() As Object
```

Note that the contents will be returned in UTF-8 format. If you would like to translate the contents into a .NET string, use the GetString() function of the System.Text.UTf8Encoding object to create a string from the byte array returned by this function.

JsonWriter.WriteField Method

The WriteField method adds content required for a writing a name/value pair representing a field within an object to the attached file or memory buffer .

Syntax

Visual Basic (usage)

```
Public Sub WriteField(name As String, value As Object)
```

WriteField is intended to simplify a common operation. It is equivalent to making a call to WriteToken with the fieldname followed by a call to WriteValue with the field value.

JsonWriter.WriteToken Method

The WriteToken method adds content required for writing a single token to the attached file or memory buffer.

Syntax

Visual Basic (usage)

```
Public Sub WriteToken(tokenType As EViewsEdx.JsonTokenType, Optional value As
Object = Nothing)
```

The following token types and values may be passed into the method.

Token Type	Value	Description
JSON_NULL	Nothing	A JSON null value
JSON_BOOL	Boolean	A Boolean value (true or false)
JSON_NUMBER	Double	A numeric value
JSON_STRING	String/Byte array	A string value
JSON_FIELDNAME	String/Byte array	An object field name
JSON_ARRAY_BEGIN	Nothing	A marker indicating the beginning of an array
JSON_ARRAY_END	Nothing	A marker indicating the end of an array
JSON_OBJECT_BEGIN	Nothing	A marker indicating the beginning of an object
JSON_OBJECT_END	Nothing	A marker indicating the end of an object
JSON_EOF	Nothing	A marker indicating that no more content will be written

JsonWriter.WriteValue Method

The WriteValue method adds content required for a data value to the attached file or memory buffer .

Syntax

Visual Basic (usage)

```
Public Sub WriteValue(value As Object)
```

WriteValue determines which tokens are required from the data type of the input value.

WriteValue may be used to write arrays with a single function call.

APPENDIX A: Attribute Formats

At a number of places in the EViews Database Extension interface, sets of attributes are passed between EViews and the database layer using a single `attr` argument containing a COM variant or .NET object.

These attribute arguments provide an open ended system to exchange loosely defined information in a way compatible with the COM type system.

Each `attr` argument represents a set of name-value pairs, where there is one pair for each attribute in the set. The number of attributes is open-ended.

There are several supported formats for how these name-value pairs are encoded inside the single argument. All of the formats are supported (and may be mixed and matched) whenever EViews receives an `attr` argument from the external database interface. When EViews is sending an `attr` argument to the external database interface, the database manager may request which format should be used by setting the `attrtype` attribute returned by the `GetAttributes` call. There is probably little difference in performance between the formats. You should choose whichever format is most convenient.

csv format

In csv format, all attributes are sent/received in a single string where the string uses a comma delimiter to separate the name value pairs. The generic format is:

"Name1=Value1, Name2=Value2, Name3=Value3, …"

If the attribute name contains a comma or equal sign, it should be surrounded by square brackets. If the attribute value contains a comma or equal sign, it should be surrounded by double quotes. Double quotes inside a value already surrounded by quotes should be escaped by double quote pairs. For example, we could have an attribute pair:

[My=Name]="a value, containing ""commas"""

strarray format

In strarray format, attributes are sent/received in a two dimensional array of strings, where there is one row for each attribute. The first column contains the attribute name and the second column contains the attribute value. The generic format is:

Name1	Value1
Name2	Value2
Name3	Value3

Since the array cell boundaries divide up the names and values, there is no need for special escape characters in this format.

objarray format

objarray format is the same as strarray format except that attributes are sent/received in a two dimensional array of objects/variants, where there is again one row for each attribute. The first column contains the attribute name and should always be a string. The second column contains the attribute value and can contain different types of data depending on the attribute. For example, start and end dates may be provided in a date type instead of using a string.

APPENDIX B: Database Attributes

Database attributes are returned to EViews by the Database manager during a call to `IDatabaseManager::GetAttributes`. The database manager attributes provide important information to EViews that tell EViews how to interact with the external database format.

The entire set of database attributes is returned in a single argument. For a discussion of the way the attributes are encoded in the argument, see Appendix A.

attrtype=csv|strarray|objarray	Used to request the format that attribute sets should be passed in to EViews. Only one value may be specified. See Appendix A for details of the meaning of the three values.
dbid=string	Default value for database id, used by EViews to fill in the database field in the Open Database dialog. May be useful in cases when the database extension supports multiple database ids, but one of them is much more commonly used than the others.
dbidlabel=string	Changes the label that will appear in the EViews graphical interface next to the field that will be sent to the external database code as `databaseId`. Default value is "Database". May be used to provide a more natural interface for people who are already familiar with particular terminology normally used with the foreign database.
dbids	Tells EViews that the database manager supports returning a list of database ids through the IDatabaseManager.GetDatabaseIds function. When this flag is provided, EViews will allow browsing the database ids in the Open Database dialog. If this attribute is not supplied, EViews will default to a file browsing interface.
description=string	Name used to describe the format within EViews dialogs (such as the Open Database dialog) and in the Database Window caption.
ext=string	For file-based databases, the primary file extension associated with the database (specified without a leading dot). When a user tries to open a file with this extension, EViews will automatically treat the file as being a database of this format without the user having to explicitly specify the format type.
extlist=string|string|string	For file-based databases consisting of multiple files, a list of all extensions associated with this file type, using "|" as the delimiter between extensions. If you only have a single file, you do not need to specify this attribute. This attribute is used by EViews when performing default implementations of entire database management functions (`dbcopy`, `dbrename`, `dbdelete`).
locking=short	Used to indicate to EViews that the database should be kept open for as short a time as possible. When `locking=short` is specified, EViews will only open a database when it is about to perform an operation (read, write, search) and close it again immediately after it is complete. If `locking=short` is not specified, EViews will keep a database open for as long as the database window is kept open on the screen. Short locking minimizes the possibility of sharing violations when multiple

	users are working on a single database. However, short locking may be less efficient in cases where opening and closing the database takes a considerable amount of time. Server-based databases for which connections from the client to the server are expensive should generally not use short locking.
login=server|user|pass|dbid	Used to indicate to EViews which fields must be specified by the user when opening a database. Fields which are not included in the login attribute will be greyed-out or omitted in dialogs inside EViews. Typically used for server-based databases.
name=string	Short name for format used in error message text and some other places. Typically only one or two words.
nocreate	Flag to indicate that the database manager cannot create new databases in this format. This will remove the database format from the New Database dialog and prevent EViews from calling the database with any `OpenCreateMode` mode other than `FileOpen`.
readonly	Flag to indicate that the database manager does not support opening databases in this format for writing. This will prevent EViews from calling the database with any `ReadWriteMode` other than `FileReadOnly` and will prevent EViews from calling any functions that require write access to the database.
search=all|attr|browser	Indicates to EViews what types of searches are supported by databases in this format. One or more values may be specified separated by "|" delimiters. EViews uses this attribute to adjust which buttons are available in the database window button bar. `All` indicates that EViews may request a list of all objects in the database. Most file-based databases should support this. Server-based databases may choose not to support this if there are too many objects to return. `Attr` indicates that EViews may call `IDatabase.SearchByAttributes` to perform a search for objects within the database that have certain attributes. `Browser` indicates that EViews may call `IDatabase.SearchByBrowser` to allow the external database to create a custom browser for the user to navigate through the database and select objects.
searchattr=attr1|attr2|attr3	List of attributes that the database supports searching over inside `IDatabase.SearchByAttributes`. Defaults to `name` which indicates that the database can filter objects by name, although even this is not necessary since the database is allowed to return extra objects to EViews and leave EViews to perform any necessary filtering. See SearchByAttributes for a discussion.
server=string	Indicates to EViews that the database is server-based, and optionally provides a default value (eg. a URL) for the server field. When this attribute is specified, EViews adds extra fields to the open database dialog so that server and login details can be input. See the `login` attribute for details on how to adjust these fields.
type=string	Value of the `type` option that can be used to select the database format in EViews commands such as `dbopen` and `dbcreate`. Should generally be limited to a single word.

APPENDIX C: Object Attributes

Object attributes are sent back and forth between EViews and the database class during calls to `ReadObject` and `WriteObject`.

The entire set of object attributes is passed in a single argument. For a discussion of the way the attributes are encoded within the argument, see Appendix A.

Object attributes contain metadata that help interpret the data values of the object (such as the frequency, start and end date of a time series) as well as variety of purely descriptive information (such as a text string containing the source of the data). EViews has a set of predefined object attributes that have particular meanings, but EViews also allows custom attributes to be attached to any object. Custom attributes can be displayed and edited within EViews but have no special meaning to EViews. Any attribute which EViews does not recognize as a predefined attribute is treated as a custom attribute, and the list of custom attributes is specific to each object. You may use custom attributes to return information in the database that does not match up with any of EViews predefined fields.

The predefined attributes use the same definitions as the main EViews database user interface. The following table contains a brief description of the attributes. Further details are available in the "EViews Databases" chapter of the EViews User's Guide.

convert_hilo	Frequency conversion option for when this object is converted into a lower frequency (contracted).
convert_lohi	Frequency conversion option for when this object is converted into a higher frequency (interpolated).
description	One line text description of the object.
display_name	Optional text used by EViews to label the object in presentation output (graphs and tables). Should be short, but may contain spaces and other characters that are illegal in EViews names.
end	Date of last observation in a time series object. May be specified in any format that EViews understands. e.g. "2000-04-01", "Apr2000", "2000Q2".
freq	Text string representing the frequency of a time series object. Uses the same notation for frequency as the EViews `create` command. Supports a wide set of common terms including: A, Annual, Q, Quarter, Quarterly, M, Month, Monthly, W, Week, Weekly. See the documentation for the Set() function of the Frequency class for more details or the main EViews documentation for a comprehensive discussion of EViews frequencies.
history	Multiline text field containing commands used to generate the object. Generated by EViews automatically based on user operations.
last_update	Timestamp of last update to the object. Changed automatically within EViews when the object is edited.
linkid	Link specification for when this object is used to create a new database link in a workfile. If this attribute is not present, EViews will use the objectId used to fetch the series as the link specification.

obs	Number of observations in a time series object.
remarks	Multiline text field containing descriptive information on the object.
source	Single line text field containing information about the source of the object.
start	Date of first observation in a time series object. May be specified in any format that EViews understands. Eg. "2000-04-01", "Apr2000", "2000Q2".
type	Single word (without spaces) indicating the type of the object. Follows the standard EViews type names including: series, scalar, vector, matrix, alpha, string, svector. See the main EViews documentation for details.
units	Single line text field containing information about the units of the object.